НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ ЗАХИСТУ ІНФОРМАЦІЇ

«На правах рукопису»
УДК _____

«До захисту допущено»
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)
“___” _грудня_ 2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 113 “Прикладна математика” _____
(код і назва)

на тему: Застосування кубічних тестрів у криптоаналізі симетричних шифрів__

Виконав: студент 6 курсу, групи ФІ-73мп
(шифр групи)

_____ Свічкарьов Іван Володимирович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.ф-м.н. Фаль О.М _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»
Фізико-технічний інститут
Кафедра математичних методів захисту інформації

Рівень вищої освіти: другий (магістерський) за освітньо–професійною програмою

Спеціальність: 113 «Прикладна математика»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедрою
_____ М.М.Савчук
(підпис) (ініціали, прізвище)

«___»__грудень__2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
_____Свічкарьов Іван Володимирович _____
(прізвище, ім'я, по батькові)

1. Тема дисертації : Застосування кубічних тестрів у криптоаналізі симетричних шифрів,
науковий керівник дисертації: к.ф.-м.н Фаль Олексій Михайлович _____ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від _____р. № _____

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження: інформаційні процеси в системах криптографічних захисту _____
_____

4. Предмет дослідження (Вхідні дані – для магістерської дисертації за освітньо–професійною програмою)
Кубічні тестери, а також сценарії та моделі кубічних атак _____
_____
_____

5. Перелік завдань, які потрібно розробити :1) Розробка платформи для реалізації різних моделей атак на існуючі шифри; _____
2) отримання результатів застосування атак на шифри: Simon, Simeck, Speck;
3) реалізація валідатора отриманих результатів _____
_____
_____

6. Орієнтовний перелік ілюстративного матеріалу : 82 аркуші, 2 додатки, 35 посилань на використані джерела, 7 рисунків та 10 таблиць _____

_____

_____

7. Орієнтовний перелік публікацій _____

_____

_____

8. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Пошук нової проблеми для подальшого вирішення	01.02 – 01.08	
2	Обрання напряму роботи	01.08 – 02.08	
3	Затвердження теми	02.08 – 02.08	
4	Аналіз поставленої проблематики	01.02 – 01.07	
5	Аналіз існуючих шифрів	01.07 – 19.11	
6	Реалізація платформи для здійснення атак	01.06 - 19.11	
7	Пошук моделей атак	01.07 - 19.11	
8	Здійснення атак на шифри: Simon, Simeck, Speck	01.07 – 19.11	
9	Отримання результатів	1.07 – 19.11	
10	Оформлення результатів	19.11 – 11.12	

Студент _____ ____Свічкарьов І.В___
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ ____Фаль О.М_____
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено наукового керівника магістерської дисертації.

РЕФЕРАТ

Дипломну роботу виконано на 81 аркуші, вона містить 2 додатки та перелік посилань на використані джерела із 35 найменувань. У роботі наведено 7 рисунків та 10 таблиць.

Метою даної дипломної роботи є огляд та застосування кубічних тестерів і кубічних атак до симетричних криптосистем, а також аналіз проблем та підходів до покращення сценаріїв атак.

Об'єктом дослідження є інформаційні процеси в системах криптографічного захисту.

Предметом дослідження є кубічні тестери, а також сценарії та моделі кубічних атак.

Проведено аналіз криптографічного метода, який пропонується для застосування до існуючих шифрів у інформаційних технологіях для оцінки стійкості, а також розроблено програмну реалізацію для застосування атак і валідатор отриманих результатів, який доступний у вигляді веб-сторінки.

Для легковісних блокових шифрів Simon32/64, Simeck32/64 було знайдено моделі кубічних атак, завдяки яким надається можливість повністю отримати секретний ключ. В результаті чого, складність знаходження секрету алгоритму шифрування зводиться до потреби наявності вибраних $2^{11.459}$ та $2^{11.554}$ відкритих текстів. Що стосується Speck32/64, то він виявився стійким до кубічних атак і не може наблизитися за ефективністю до інших типів атак.

КУБІЧНІ ТЕСТЕРИ, КУБІЧНІ АТАКИ, АТАКА ЗА ПОБІЧНИМИ КАНАЛАМИ, SIMON, SIMECK, SPECK, ВАГА ХЕМІНГА, БІТ ІНФОРМАЦІЇ

РЕФЕРАТ

Дипломную работу выполнено на 81 листе, она содержит 2 приложения и перечень ссылок на использованные источники из 35 наименований. В работе приведены 7 рисунков и 10 таблиц.

Целью данной работы является обзор и применение кубических тестеров и кубических атак к симметричным криптосистемам, а также анализ проблем и подходов к улучшению сценариев атак.

Объектом исследования являются информационные процесы в системах криптографической защиты.

Предметом исследования являются кубические тестеры, а также сценарии и модели кубических атак.

Проведен анализ криптографического метода, который предлагается для применения в существующих шифрах в информационных технологиях для оценки устойчивости, а также разработана программная реализация для применения атак и валидатор полученых результатов, который доступен в виде веб-страницы.

Для легковесных блочных шифров Simon32/64, Simeck32/64 были найдены модели кубических атак, благодаря которым предоставляется возможность полностью получить секретный ключ. В результате чего, сложность нахождения секрета алгоритма шифрования сводится к необходимости наличия выбранных $2^{11.459}$ и $2^{11.554}$ открытых текстов. Что касается Speck32/64, то он оказался устойчивым к кубическим атакам и не может приблизиться по эффективности к другим типам атак.

КУБИЧЕСКИЕ ТЕСТЕРЫ, КУБИЧЕСКИЕ АТАКИ, АТАКА ПО ПОБОЧНИМ КАНАЛАМ, SIMON, SIMECK, SPECK, ВЕС ХЭММИНГА, БИТ ИНФОРМАЦИИ

ABSTRACT

The thesis is presented in 81 pages. It contains 2 appendixes and bibliography of 35 references. 7 figures and 10 tables are given in the thesis.

The purpose of this paper is to review and apply cube testers and cube attacks to symmetric cryptosystems, as well as analyze problems and approaches to improving attack scenarios.

The object is information processes in cryptographic protection systems.

The subject is cube testers, as well as scenarios and models of cube attacks.

In this thesis, cryptographic method are analyzed, which is proposed for application in existing ciphers in information technology for stability assessment, and also developed a software implementation for the application of attacks and a validator of the results, which is available as a web page.

For the lightweight block ciphers Simon32/64, Simeck32/64, models of cube attacks were found, thanks to which it is possible to fully obtain the secret key. As a result, the difficulty of finding the secret of the encryption algorithm is reduced to the need of having selected $2^{11.459}$ and $2^{11.554}$ plaintexts. As for Speck32/64, it turned out to be resistant to cube attacks and can't come close in efficiency to other types of attacks.

CUBE TESTERS, CUBE ATTACK, SIDE CHANNEL ATTACK, SIMON, SIMECK, SPECK, HAMMING WEIGHT, BIT OF INFORMATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

$\mathbb{GF}(q)$ — поле Галуа порядку q

IV — вектор ініціалізації

$SCCA$ — кубічна атака за побічним каналом

HW — вага Хемінга

$\oplus$ — додавання за модулем 2

ARX — *від англ. add, rotate, xor*

LSB — *від англ. least significant bit*

MSB — *від англ. most significant bit*

$\ll, \gg$ — зсув біт ліворуч та праворуч відповідно

$\lll, \ggg$ — циклічні зсуви ліворуч та праворуч, відповідно

ВСТУП

Актуальність дослідження. На сьогоднішній день розвиток інформаційних систем не зупиняється, через що інформаційна безпека повинна розвиватися із дня в день. На даний процес покладені великі зусилля та залучена широка спільнота.

Дослідження алгоритмів шифрування та криптографічних примітивів призводить до їхнього покращення, а навіть до кардинального перегляду методів побудови. Значний внесок у розвиток криптології на початку 90-х років зробили основоположники диференціального криптоаналізу Біхам та Шамір, а також японський криптограф Міцуру Мацуї, який розробив лінійний криптоаналіз.

У 2008 році з'являється новий метод сучасного криптоаналізу симетричної криптографії – кубічна атака, представлена Дінуром та Шаміром [2]. Вона поєднує в собі клас лінійних, диференціальних, алгебраїчних і кореляційних атак. Її перевага в тому, що кубічна атака може бути застосована до будь-якого блокового, потокового шифру або MAC-коду, про внутрішню структуру яких нічого невідомо.

Мета і завдання дослідження. Метою роботи є огляд та застосування кубічних тестерів і кубічних атак до симетричних криптосистем, а також аналіз проблем та підходів до покращення сценаріїв атак.

Досягнення поставленої мети передбачає вирішення таких завдань, які були виконані у роботі:

1) аналіз кубічних тестерів;
2) аналіз запропонованої атаки Дінура та Шаміра;
3) аналіз динамічних кубічних атак;
4) аналіз різних моделей атак на симетричні криптосистеми;
5) аналіз практичних проблем при застосуванні даного типу атак;
6) розробка та реалізація платформи запропонованих моделей атак;

7) розробка валідатора результатів запропонованих моделей атак;

8) застосування атак до конкретних алгоритмів шифрування та отримання даних для відновлення ключа.

Об'єктом дослідження є інформаційні процеси в системах криптографічного захисту.

Предметом дослідження є кубічні тестери, а також сценарії та моделі кубічних атак.

При розв'язанні поставлених завдань використовувались такі *методи дослідження*: методи комбінаторного аналізу, теорії складності алгоритмів, теорії імовірностей, методи комп'ютерного моделювання та проектування.

Наукова новизна отриманих результатів полягає у здійсненні криптоаналізу блокових легковісних шифрів Simon, Simeck, Speck на основі інформації за побічними каналами, який дає повністю відновлений ключ для перших двох.

Практичне значення результатів полягає у отриманні інформації за побічними каналами, яка є основою для оцінювання стійкості алгоритмів шифрування.

1 КУБІЧНІ ТЕСТЕРИ ТА КУБІЧНІ АТАКИ

У даному розділі розглядаються необхідні теоретичні відомості по кубічним атакам та кубічним тестерам. Загальні положення та терміни викладено згідно [1, 2, 5, 6, 7]

1.1 Вступ

Головною метою криптоаналізу є розкриття секрету криптосистеми, порушення конфіденційності і цілісності інформації, а також виявлення уразливостей криптографічних примітивів. Стійкість криптосистеми не повинна спиратися на секретність її будови, алгоритму шифрування, а повинна ґрунтуватися на секретності ключа при надійному алгоритмі шифрування і достатньому розмірі ключа.

Добре спроектований шифр повинен бути стійким до усіх відомих атак, включаючи атаки з розпізнавачами (distinguishing attacks) та з відновленням ключа (key recovery attacks). Ці два типи атак тісно пов'язані, оскільки у багатьох випадках одна може бути розширена до іншої. Прикладами може виступати диференціальний та лінійний криптоаналіз, в яких атакуючий створює розпізнавач для певного числа раундів ітеративного блокового шифру (зазвичай це атаки на ключ останнього раунду). Ці два типи криптоаналізу відносяться до атак на основі вибраних відкритих текстах.

Розпізнавач може бути легко використаний, щоб перевірити припущення:

– при правильній гіпотезі, частково розшифровані шифротексти, як очікується, будуть виявляти невипадкові властивості;

– при неправильній – це насправді еквівалентно додаванню додаткового раунду шифрування, а отже, розшифровані шифротексти ведуть себя випадковим чином.

Це загальна схема сатистичної атаки на ключ останнього раунду. Такі розпізнавачі не застосовні до потокових шифрів, де відсутнє часткове розшифрування, але застосовні до ітеративних блокових шифрів, що може використовуватися у динамічних кубічних атаках [3, 4], які будуть розглянуті у підрозділі 1.4.

1.2 Кубічні атаки

Кубічні атаки [2] є свого роду алгебраїчними атаками, в основі яких лежать атаки із вибраними відкритими текстами (тобто коли криптоаналітику надається доступ не тільки до шифротекстів, але і до відповідних відкритих текстів). Для її застосування необхідно мати лише доступ до чорного ящика у який зашита будь-яка симетрична криптосистема.

Криптосистема може бути описана у вигляді полінома $p(\vec{v}, \vec{k})$ над $\mathbb{GF}(2)$, де $\vec{v} = (v_0, \ldots, v_{m-1})$ представляє собою відкриті параметри (вектор ініціалізації (initialization vector) або блок відкритого тексту для потокових та блокових шифрів відповідно), а $\vec{k} = (k_0, \ldots, k_{n-1})$ – секретні змінні, які є бітами ключа.

Загалом, кубічна атака виконується у два етапи: передобчислювальний (preprocessig phase) та онлайн етап (online phase).

Для успішної атаки необхідно накопичити достатню кількість незалежних рівнянь від секретних змінних на передобчислювальному етапі, потім отримати їх праву частину на онлайн етапі та вирішити сформовану систему для отримання ключових біт ефективніше за повний перебір (тобто за часову складніть $< 2^n$). Накопичувати можливо як лінійні, так і квадратичні рівняння, але це вже призводить до вирішення системи нелінійних рівнянь, що значно складніше за лінійну систему.

Для довільної множини $I \subseteq \{0, \ldots, m-1\}$, поліном $p(\vec{v}, \vec{k})$ може бути представлений у наступному вигляді:

$$p(\vec{v}, \vec{k}) = t_I \cdot p_{S(I)}(\vec{k}) \oplus q(\vec{v}, \vec{k}), \tag{1.1}$$

де $t_I = \prod_{i \in I} v_i$, а $p_{S(I)}$ – називають *суперполіномом (від* англ. superpoly*)* для множини I.

Визначення 1.1. Терм t_I називають макстермом (від *англ. maxterm*), якщо $deg(p_{S(I)}) \equiv 1$, тобто $p_{S(I)}$ лінійний і не константа.

Визначення 1.2. Будь-яка підмножина I розміру l, яка визначає індекси куба, формує l–розмірний булевий куб C_I ($|C_I| = 2^l$, де у кожному векторі множини C_I підвектор який складається із бітів номери якого належать множині I приймають усі можливі значення, а усі інші біти з номерами $i \in I$ приймають статичне значення – 0 або 1), а будь-який вектор $\vec{w} \in C_I$ – новий поліном $p(\vec{w}, \vec{k})$ із $n - l$ змінними. Підсумовуючи такий поліном над усіма можливими векторами із C_I, отримаємо в кінцевому підсумку новий поліном, який позначається через $p_I(\vec{k}) \triangleq \bigoplus_{w \in C_I} p(\vec{w}, \vec{k})$.

Приклад 1.1. Якщо обрати множину $I = \{0,1,2\}$ та $m = 5$, тоді 5-розмірний булевий куб буде таким:

$$C_I = \{(0,0,0,0,0), (0,0,1,0,0), \ldots, (1,1,0,0,0), (1,1,1,0,0)\}.$$

Теорема 1.1. *Для довільного полінома* $p(\vec{v},\vec{k})$ *і підмножини* I, $p_{S(I)}(\vec{k}) \equiv p_I(\vec{k})\ mod\ 2$ *(доведення див. у [2]).*

Нехай $p_{S(I)}(\vec{k}) : \{0,1\}^n \to \{0,1\}, n > 0$, алгебраїчна нормальна форма (АНФ) якого представляється наступним чином:

$$p_{S(I)}(\vec{k}) = \bigoplus_{i=0}^{2^n-1} a_{i_0 \ldots i_{n-1}} k_0^{i_0} k_1^{i_1} \ldots k_{n-1}^{i^{n-1}}, \tag{1.2}$$

де $a_{i_0 \ldots i_{n-1}} \sim a_i$ – коефіцієнт при відповідному мономі, а i_j дорівнює j-тому біту двійкового представлення числа i,.

Пошук АНФ можна виконувати за наступним алгоритмом (що є модифікацією так званого перетворення Мебіуса). Нехай k_i – вектор, в якому i-тий біт дорівнює 1, а всі інші – 0. Тоді, як неважко помітити, виконуються такі співвідношення:

$$\begin{aligned}
a_0 &= p_{S(I)}(\vec{0}) \\
a_i &= p_{S(I)}(\vec{k_i}) \oplus a_0 \\
a_{ij} &= p_{S(I)}(\vec{k_i} \oplus \vec{k_j}) \oplus a_i \oplus a_j \oplus a_0 \\
a_{ijk} &= p_{S(I)}(\vec{k_i} \oplus \vec{k_j} \oplus \vec{k_k}) \oplus a_{ij} \oplus a_{ik} \oplus a_{jk} \oplus a_i \oplus a_j \oplus a_k \oplus a_0
\end{aligned}$$

і так далі, тобто коефіцієнти поліному обчислюються послідовно, від молодших степенів до старших.

В основі атаки лежить спостережння, що якщо поліном $p(\vec{v},\vec{k})$ має степінь d, то підсумовуючи його над усіма векторами із $(d-i)$–розмірного куба, де $i \in \overline{1,d-1}$ веде до утворення лінійних, квадратичних і так далі суперполіномів.

Далі наводиться приклад, який демонструє попередньо описані відомості.

Приклад 1.2. Нехай $p(\vec{v},\vec{k})$ має наступне представлення:

$$\begin{aligned}
p\,(v_0,v_1,v_2,k_0,k_1,k_2) &= v_0v_1k_0 \oplus v_0v_1k_2 \oplus v_1k_2 \oplus v_1k_1 \oplus v_1k_0 \oplus v_0v_1 \\
&\oplus\ v_2v_1k_1k_0 \oplus v_2v_1k_0k_2 \oplus v_2v_1 \oplus v_1 \oplus k_1 \oplus 1
\end{aligned}$$

Обираючи різні множини I можна отримати лінійні та квадратичні суперполіноми:

1) $I = \{0,1\}$ дає лінійний суперполіном.

$p(\vec{v},\vec{k})$ можна подати у вигляді (1.1), де

$$\begin{aligned} t_I &= v_0 v_1 \\ p_{S(I)}(\vec{k}) &= k_0 \oplus k_2 \oplus 1 \\ q(\vec{v},\vec{k}) &= v_1k_2 \oplus v_1k_1 \oplus v_1k_0 \oplus v_2v_1k_1k_0 \oplus v_2v_1k_0k_2 \oplus v_2v_1 \oplus v_1 \oplus k_1 \oplus 1 \end{aligned}$$

Далі знаходиться p_I згідно визначення (1.2) і перевіряється те, що теорема (1.1) виконується. Знайдемо p_I згідно визначення (1.2) і переконаємося, що теорема (1.1) виконується.

$$\begin{aligned} p_I &= 0 \cdot 0(k_0 \oplus k_2 \oplus 1) \oplus 0 \cdot 1(k_0 \oplus k_2 \oplus 1) \\ &\oplus 1 \cdot 0(k_0 \oplus k_2 \oplus 1) \oplus 1 \cdot 1(k_0 \oplus k_2 \oplus 1) \\ &\oplus (0 \cdot k_2 \oplus 0 \cdot k_1 \oplus 0 \cdot k_0 \oplus 0 \cdot 0 \cdot k_1k_0 \oplus 0 \cdot 0 \cdot k_0k_2 \oplus 0 \cdot 0 \oplus 0 \oplus k_1 \oplus 1) \\ &\oplus (1 \cdot k_2 \oplus 1 \cdot k_1 \oplus 1 \cdot k_0 \oplus 0 \cdot 1 \cdot k_1k_0 \oplus 0 \cdot 1 \cdot k_0k_2 \oplus 0 \cdot 1 \oplus 1 \oplus k_1 \oplus 1) \\ &\oplus (0 \cdot k_2 \oplus 0 \cdot k_1 \oplus 0 \cdot k_0 \oplus 0 \cdot 0 \cdot k_1k_0 \oplus 0 \cdot 0 \cdot k_0k_2 \oplus 0 \cdot 0 \oplus 0 \oplus k_1 \oplus 1) \\ &\oplus (1 \cdot k_2 \oplus 1 \cdot k_1 \oplus 1 \cdot k_0 \oplus 0 \cdot 1 \cdot k_1k_0 \oplus 0 \cdot 1 \cdot k_0k_2 \oplus 0 \cdot 1 \oplus 1 \oplus k_1 \oplus 1) \end{aligned}$$

$$\Rightarrow p_{S(I)}(\vec{k_i}) = p_I(\vec{k}) \, mod \, 2 = k_0 \oplus k_2 \oplus 1$$

2) $I = \{1,2\}$ дає квадратичний суперполіном.

$$p_{S(I)}(\vec{k}) = \bigoplus_{(w_1,w_2)\in\{0,1\}^2} p(0,w_1,w_2,k_0,k_1,k_2) = k_0k_1 \oplus k_0k_2 \oplus 1$$

3) $I = \{0,1,2\}$ дає нульовий суперполіном.

$$p_{S(I)}(\vec{k}) = 0$$

$$a_{2^3-1} = \bigoplus_{(k_0,k_1,k_2)\in\{0,1\}^3} p_{S(I)}(k_0,k_1,k_2) = 0$$

Загалом, звичайну кубічну атаку можна подати наступним алгоритмом (1.1).

Алгоритм 1.1. Кубічна атака

Вхід: q – максимальная степінь суперполіному $p_{S(I)}(\vec{k})$;
$\{N_i\}_{1<i\leq q}$ – число тестів лінійності/нелінійності;
Вихід: K – множина бітів ключа;
CI – множина підмножин I (індексів кубічних змінних)

1 **до тих пір, поки** *не знайдено достатню кількість незалежних суперполіномів* **виконати**
2 Вибрати випадковим чином розмірність куба $l : 1 \leq l < m$, підмножину $|I| = l$ та перевірити $1 \leq deg(p_{S(I)}(\vec{k})) \leq q$ відповідну кількість разів N_i(алгоритми наведені у [13, 14]);
3 **якщо** $p_{S(I)}$ *пройшов тест* $deg(p_{S(I)}(\vec{k})) \leq q$ **тоді**
4 $CI \leftarrow I$;
5 відновити символьний вигляд $p_{S(I)}(\vec{k})$ (алгоритми наведені у [1, 2, 6]);
6 **інакше**
7 обрати іншу розмірність k;
8 **кінець умови**
9 **кінець циклу**
10 Знайти праву частину отриманих рівнянь, підсумовуючи $p(\vec{v}, \vec{k})$ над знайденими кубами і фіксованим ключем $\vec{k}$ який потрібно знайти;
11 Вирішити вихідну систему та записати рішення у K;
12 **повернути** K, CI

У подальших атаках кроки 1-10 пропускаються (тобто передобчислювальний етап), замість цього використовується вже відома множина CI та виконуються тільки кроки 10-11 (онлайн етап).

Теоретична складність атаки

Розглянемо поліном степіня d із n секретними та m відкритими вхідними бітами. Складність обчислення значень n лінійних суперполіномів оцінюється щонайбільше $2^{d-1}n$ запитами до оракула. Складність рішення системи лінійних рівняння $\mathcal{O}(n^2)$ [1].

Таким чином, загальна складність онлайн-атаки $\mathcal{O}(2^{d-1}n) + \mathcal{O}(n^2)$.

Що стосується етапу попередньої обробки, він складається з оцінки кубів, кожна з яких оцінюється $\mathcal{O}(2^{d-1})$.

Загальна складність оцінки нелінійних рівнянь степіня D, використовуючи алгоритми із [14], наступна:

$$\kappa n 2^{d-D+1} + \sum_{i=0}^{D} 2^{d-D+i} C_N^i$$

де κ–кількість тестів для знаходження секретних змінних, а N–кількість таких секретних змінних. Для більшої впевненності у правильності роботи алгоритму, $\kappa = 300$.

В нашому випадку D буде обиратися не вище другого степеня, тож складність відновлення суперполінома другого степіня, складає:

$$\kappa n 2^{d-1} + 2^{d-2} + 2^{d-1}N + 2^d \, \frac{N(N-1)}{2}$$

Слід зазначити, що значення d, як правило, не відоме до виконання атаки, так що верхня межа існує, але зловмисник не обов'язково знає яка вона.

Покращення кубічних атак і тестерів буде описано у наступних розділах, а робочий код можна знайти за посиланнями [21, 22].

[1] За умови, що матриця лінійних рівнянь невироджена, яка може бути доповнена ще кількома суперполіномами (superpolys), якщо це необхідно.

1.3 Кубічні тестери

На відміну від кубічних атак, кубічні тестери [1] зазвичай є атаками із розпізнавачами, а не атаками із відновленням ключа. Кубічні тестери об'єднуються із кубічними атаками за для ефективного встановлення властивостей поліномів і можуть бути використані у знаходженні розпізнавальників або щоб виявити випадковість у криптографічних примітивах. Такі тестери є гнучкими атакам, вони не вимагають, щоб атакована функція мала низьку степінь.

Далі у данному підрозділі для спрошення позначень буде ігноруватися позначення між відкритими та секретними змінними, та визначатимуться вектором $\vec{x} = (x_0, \ldots, x_{l-1})$, де $l = m+n$. А мастер поліном буде подаватися у наступному вигляді:

$$f(\vec{x}) = t_I \cdot p_{S(I)}(...) \oplus q(\vec{x}).$$

Також усі змінні поділяються на 3 множини:

– кубічні змінні (cube variables (CV))

– змінні суперполінома (superpoly variables (SV))

– інші змінні

Третя множина може мати нульову потужність якщо ($|CV|+|SV| = l$).

Кубічні тестери виявляють випадковість, перевіряючи властивості суперполіномів: неформально це пояснюється так, що як тільки суперполіном має якусь несподівану властивість, то він визначається як невипадковий.

Такими властивостями можуть бути :

– ***Збалансованість.*** Дисбалансом булевої функції $f : \{0,1\}^l \to \{0,1\}$ називається величина

$$disb(f) = \left| \sum_{x \in \{0,1\}^l} \mathbb{1}\{f(x) = 0\} - \sum_{x \in \{0,1\}^l} \mathbb{1}\{f(x) = 1\} \right|$$

Дисбаланс показує, наскільки імовірність одержати одне значення на виході булевої функції більше за імовірність одержати інше.

Очікується, що випадкова функція буде містити таку ж кількість нулів і одиниць, що міститься у таблиці істинності. Сильно незбалансовану булеву функція можна відрізнити від випадкової, перевіряючи її дисбаланс. Як правило, не запитується уся таблиця істинності, а лише її випадкова вибірка.

Якщо змінні поділяються тільки на CV та SV, де $|CV| = C$, а $|SV| = S$, то даний тест можна описати у вигляді алгоритму (1.2).

Алгоритм 1.2. Тест на збалансованість

Вхід: $CV = \{x_0, \ldots, x_{C-1}\}$ та $SV = \{x_C, \ldots x_{l-1}\}$ – множини відповідних змінних;
$N < 2^S$ – число тестів;
$\mathcal{D}$ – правило перевірки гіпотези;

Вихід: $\mathcal{D}(c) \in \{0,1\}$;

1 $c \leftarrow 0$;
2 **цикл** *від* $t = 1$ ***до*** N **виконати**
3 обрати випадково і рівноймовірно $(x_C, \ldots x_{l-1})$;
4 $c \mathrel{+}= \bigoplus_{(x_0,\ldots,x_{C-1}) \in \{0,1\}^C} f(x_0, \ldots, x_{l-1})$
5 **кінець циклу**
6 **повернути** $\mathcal{D}(c)$

Складність цього імовірносного тесту $N2^C$, ну а якщо брати детерміністичний, то це 2^l.

– ***Константність.***

Частковий випадок збалансованості, коли суперполіном $p_{S(I)}$ максимально незбалансований (тобто у таблиці істинності одні одиниці або нулі). Це досягається коли $f(\vec{x})$ має степінь C.

– ***Низька степінь.***

Суперполіном випадкової функції має степінь $\geq S-1$ із великою вірогідністю. Оцінити степінь булевої функції можна використовуючи алгоритми описані у [13, 14, 27])

Приклад 1.3. $deg(p_{S(I)}) = 3$

$$f(x_0,\ldots,x_{l-1}) = x_0x_1(x_2x_3 \oplus x_4x_5x_7 \oplus x_{10}) \oplus q(x_0,\ldots,x_{l-1})$$

– ***Наявність лінійних змінних.***

Це частковий випадок тесту на перевірку полінома на лінійність, в якому спочатку знаходять індекси з яких складається поліном, а потім перевіряють наявність змінної у ньому за алгоритмом (1.3).

Алгоритм 1.3. Тест на наявність лінійної змінної

Вхід: x_i – змінна яка перевіряється;
$N \leq 2^{S-1}$ – число тестів;
Вихід: відповідь – лінійна/нелінійна;

цикл *від* $t = 1$ ***до*** N **виконати**
  обрати випадково і рівноймовірно $(x_0,\ldots,x_{i-1},\#,x_{i+1},\ldots,x_S)$;
  якщо
  $p_{S(I)}(x_0,\ldots,x_{i-1},0,x_{i+1},\ldots,x_{l-1}) = p_{S(I)}(x_0,\ldots,x_{i-1},1,x_{i+1},\ldots,x_S)$
  тоді
    нелінійна;
  кінець умови
кінець циклу
повернути *лінійна*

Знайти множину індексів можна, розрахуваваши усі коефіцієнти a_i у поліномі (1.2).

Продемонстрований тест (1.3) являється ймовірнісним, який підтверджує лінійність змінної x_i. Він є одностороннім, тобто у ньому відсутні помилки першого роду: якщо змінна – лінійна, то тест завжди поверне, що вона лінійна. Однак якщо вона – нелінійна, то для кожного можливого вибору вектора $(x_0, \ldots, x_{i-1}, \#, x_{i+1}, \ldots, x_S)$ тест із імовірністю $\frac{1}{2}$ поверне помилкову відповідь. Використовуючи N прогонів, помилка зменшується до 2^{-N}.

Приклад 1.4. x_4 – лінійна змінна

$$f(x_0, \ldots, x_{l-1}) = x_0 x_1 x_2 x_3 (x_4 \oplus g(x_5, \ldots, x_{l-1})) \oplus q(x_0, \ldots, x_{l-1})$$

– ***Наявність нейтральних змінних.***

Аналогічна техніка із попереднім тестом, але перевіряє нейтральність змінної у суперполіномі.

Алгоритм 1.4. Тест на нейтральність

Вхід: x_i – змінна яка перевіряється;
$N \leq 2^{S-1}$ – число тестів;
Вихід: відповідь – нейтральна/не нейтральна;

цикл *від* $t = 1$ ***до*** N **виконати**
  обрати випадково і рівноймовірно $(x_0, \ldots, x_{i-1}, \#, x_{i+1}, \ldots, x_S)$;
  якщо
  $p_{S(I)}(x_0, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_S) \neq p_{S(I)}(x_0, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_S)$
  тоді
    не нейтральна;
  кінець умови
кінець циклу
повернути *нейральна*

Приклад 1.5. x_4 – нейтральна змінна

$$f(x_0, \ldots, x_{l-1}) = x_0 x_1 x_2 x_3 \cdot g(x_5, \ldots, x_{l-1}) \oplus q(x_0, \ldots, x_{l-1})$$

1.4 Динамічні кубічні атаки

Атака на потокові шифри

Вперше динимачні кубічні атаки були представлені у роботі Дінура і Шаміра на потоковий шифр Grain-128 [3]. Дана атака представляє модифіковану версію кубічних атак, яка відновлює секретний ключ криптосистеми шляхом використання розпізнавачів, які надаються кубічними тестерами.

Головне спостереження яке тут використовується, полягає у тому, що, коли дані криптосистеми недостатньо замішуються, стійкість таких шифрів до кубічних тестерів зазвичай залежить від кількох нелінійних операцій, які виконуються на останніх етапах шифрування (у раундовій функції для блокових шифрів). Вони призводять до утворення доданків відносно великого степіня у алгебраїчній нормальній формі вихідної функції. Якщо вдається спростити таке представлення проміжного стану шифрування, наприклад, обнуляючи якийсь біт входу, то степінь поліному може бути набагато нижчим, що робить його більш уразливим для кубічних тестерів. В динамічній атаці для потокових шифрів треба аналізувати шифр, для того щоб знайти такий біт проміжного стану, який обнулить частину поліному і призведе до спрощення вихідної функції. Такі біти називають динамічними змінними (dynamic variables).

Так як проміжні біти шифрування, як правило, залежать також і від деяких ключових бітів, їх необхідно вгадувати, тобто робити якесь припущення стосовно значення цих біт, щоб потім застосувати атаку. До кожного припущення застосовуються кубічні тестери, для того щоб відрізнити функцію від випадкової. Для правильної гіпотези, частина доданків у АНФ проміжного стану шифрування обнуляється, а кубічні тестери, швидше за все, виявлять невипадкові властивісті на виході. З іншого боку, для великої частини невірних гіпотез, кубічні тестери навряд

чи виявлять таку властивість. Таким чином, можна ефективно відкинути неправильні припущення стосовно ключових біт і відновити частину секретного ключа.

Що кубічні тестери, що динамічні кубічні атаки, обидва підсумовують функцію за деяким кубом, де кубічні змінні вибираються серед множини відкритих змінних. Однах якщо перші фіксують значення усіх публічних змінних, які відмінні від кубічних змінних у 0 або 1 (такі змінні називають стачними), то другі – ні. Замість цього цим змінним, які називається динамічними, призначається функція, яка залежить від деяких кубічних змінних та деяких секретних змінних. Кожна така функція ретельно підбирається, для того, щоб обнулити частину вихідної функції, тим самим посилюючи гіпотезу про невипадковість, яка перевіряється кубічними тестерами.

Динамічні кубічні атаки дозволяють безпосередньо отримувати інформацію про секретний ключ без вирішення систем алгебраїчних рівняннь, як це було у звичайних кубічних атаках. Крім того, ретельний вибір динамічних змінних покращує складність отримання розпізнавачів за допомогою кубічних тестерів. Недоліком цієї атаки у порівнянні із кубічними атаками є те, що для перших вимагається складний етап аналізу внутрішньої структури шифру, коли другі такого не вимагають.

Далі наводиться приклад, який демонструє ідею попередньо описаної атаки.

Приклад 1.6. Нехай мастер поліном $p(\vec{v},\vec{k})$ представляється наступним чином:

$$p(\vec{v},\vec{k}) = p_1(\vec{v},\vec{k})p_2(\vec{v},\vec{k}) \oplus p_3(\vec{v},\vec{k}),$$

де $\vec{v} = (v_1,v_2,v_3,v_4,v_5)$, $\vec{k} = (k_1,k_2,k_3,k_4,k_5)$.

А поліноми $p_i(\vec{v},\vec{k}), i = \overline{1,3}$, маються такий вигляд:

$$\begin{aligned} p_1(\vec{v},\vec{k}) &= v_2v_3k_1k_2k_3 \oplus v_3v_4k_1k_3 \oplus v_2k_1 \oplus v_5k_1 \oplus v_1 \oplus v_2 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_5 \oplus 1 \\ p_2(\vec{v},\vec{k}) &= random \\ p_3(\vec{v},\vec{k}) &= v_1v_4k_3k_4 \oplus v_2k_2k_3 \oplus v_3k_1k_4 \oplus v_4k_2k_4 \oplus v_5k_3k_5 \oplus k_1k_2k_4 \oplus v_1 \oplus k_2 \oplus k_4 \end{aligned}$$

Загалом простежується, що $p(\vec{v},\vec{k})$ веде себе як випадкова функція, та здається, що вона стійка до кубічних тестерів. Однак якщо вдасться обнулити поліном $p_1(\vec{v},\vec{k})$, отримаємо що $p(\vec{v},\vec{k}) = p_3(\vec{v},\vec{k})$. Так як $p_3(\vec{v},\vec{k})$ доволі проста функція, вона може легко бути відрізнена від випадкової.

Якщо покласти $v_1 = v_2v_3k_1k_2k_3 \oplus v_2k_1 \oplus v_5k_1 \oplus v_2 \oplus k_2 \oplus k_3 \oplus k_4 \oplus k_5 \oplus 1$, а $v_4 = 0$, можна досягти такого результату, а саме щоб $p(\vec{v},\vec{k}) = p_3(\vec{v},\vec{k})$:

$$\begin{aligned} p(\vec{v},\vec{k}) &= v_2k_2k_3 \oplus v_3k_1k_4 \oplus v_5k_3k_5 \oplus k_1k_2k_4 \oplus v_2v_3k_1k_2k_3 \oplus v_2k_1 \oplus v_5k_1 \\ &\oplus v_2 \oplus k_3 \oplus k_5 \oplus 1. \end{aligned}$$

Тут v_1–визначається як динамічна змінна. Далі обравши за кубічні змінні (v_2,v_3,v_5) або будь-який інший підкуб із цими змінними можна легко відрізнити функцію $p(\vec{v},\vec{k})$ від випадкової.

Однак на практиці неможливо от так зразу встановити значення змінної v_1, так як вона залежить від ключових бітів. Доводиться вгадувати вирази $k_1k_2k_3, k_1$ та $k_2 \oplus k_3 \oplus k_4 \oplus k_5 \oplus 1$ та перевіряти правильність своїх припущень.

Атака на блокові шифри

Механізм, за допомогою якого отримуються динамічні змінни досить різний для потокових та блокових шифрів. Основний підхід атак на потокові шифри є використання рекурсивного опису вихідної функції

шифру для того, щоб спростити/обнулити деякі відповідні проміжні змінні, що призводить до спрощення її алгебраїчного представлення. Цей процес вимагає складного та кропотливого процесу, що не може бути повністю автоматизованим і передбачає ручну роботу по аналізу шифру.

Проте для блокових шифрів [4] ситуація відрізняється, так як на кожному раунді доступна ключозалежна функція, яка дає змогу криптоаналітику виконувати часткове шифрування/розшифрування у будь-якому місці шифру, та перевіряти правильність своїх гіпотез стосовно раундових ключів/біт ключа. Ця техніка недоступна у потокових шифрах.

Незважаючи на дуже складну ручну процедуру динамічних кубічних атак на потокові шифри, атака на блокові шифри може бути повністю автоматизована. Ідея полягає у тому щоб поставити r_c-раундового кубічного тестера (розпізнавача) кудись всередину шифру. Таким чином кубічні змінні вибираються не серед публічних змінних $\vec{v}$, а серед бітів проміжного стану шифрування. Потім атака протягується на r_u раундів вниз, та r_l раундів вгору після розпізнавача. Обидва розширення полягають у вгадуванні бітів ключа.

Як і у звичайних кубічних атак, динамічні атаки на блокові шифри теж поділяються на два етапи: етап передобчислення та онлайн етап. На першому етапі зловмисник намагається знайти кубічного тестера для підмножини ключових бітів, а також достатньо багато відповідних відкритих текстів від кубічних змінних та знайденої підмножини бітів ключа. В онлайн фазі, для кожного припущення щодо бітів ключа, атакуючий запускає кубічного тестера відповідними запитами до оракула та перевіряє щоб пара (P, C) відповідала йому. Якщо це так, то припущення відносно ключа правильне і він є кандидатом у правильний ключ, який буде потім перевірено.

Висновки до розділу 1

У даному розділі проведено аналіз, відбір та узагальнення теоретичних відомостей, необхідних для подальшого дослідження. На основі проведеного аналізу встановлено, що існуючий апарат, який представлений кубічною атакою, дозволяє отримати секретні параметри, навіть, для моделі чорного ящику за допомогою атак із вибраними відкритими текстами на блокові та потокові шифри.

Кубічні тестери ж є атаками із розпізнавачами. Перевагами їх є:

– відсутність складного та довго здійснюваного передобчислювального етапу;

– не потребують малого степеня функції.

Недоліками ж є:

– дають тільки розпізнавач, а не відновлення ключа;

– знаходить лише можливість атаки на функцію.

Якщо казати про динамічні кубічні атаки, то це більш загальна версія кубічних атак, в якій використовуються кубічні тестери для того, щоб визначити, чи є припущення стосовно підмножини ключових бітів – правильною чи ні. Та на відміну від класичної кубічної атаки, вони використовують інформацію про структуру шифру. Отже, завдяки цьому можна потенційно досягти кращих результатів. Однак є різниця, застосовуючи такую атаку на потокові та блокові шифри. Якщо у перших неможливо автоматизувати весь процес, то у других така перевага є.

2 АНАЛІЗ ЕФЕКТИВНОСТІ ЗАСТОСУВАННЯ КУБІЧНИХ ТЕСТЕРІВ ТА КУБІЧНИХ АТАК

Даний розділ присвячується аналізу практичного застосування кубічних атак та тестерів до симетричних шифрів. Через інтерес застосування цієї техніки до блокових шифрів, далі будуть розглянуті суто вони.

2.1 Оцінка ефективності кубічних атак

Якщо казати про блокові шифри, де використовують достатню кількість раундів, щоб степінь поліному зростала експоненційно, то пошук кубів у такому випадку стає дуже складним і наразі відсутній ефективний пошук. Тому для цього, як мінімум, намагаються використовувати не повний перебір, а рандомний, який завдяки зміні розміру кубу дає змогу, на відміну від повного перебору, знаходити куби трохи ефективніше. Якщо ж відома степінь d, то знаходження куба може стати легче, через пошук у меншій множині, але це переважно стосується початкових раундів тих шифрів, які мають просту раундову функцію. Проте знання степіня або навіть повний вигляд шифруючої функції не гарантнує, що це суттєво дає перевагу атакуючому і її можна легко зламати.

Коли аналітик працює у моделі білого ящику, тобто знає структуру алгоритму шифрування, то є можливість оцінити фактичну степінь або верхню границю степіня d на довільному раунді. Так як вихід шифруючого

перетворення розглядається у $\mathbb{GF}(2)$, то

$$\begin{aligned} d(X^{(i)}) &= \max_{x_i \in X^{(i)}} d(x_i) \leq m + n \\ d(p_{S(I)}) &\leq n \end{aligned}$$

де n – розмір ключа у бітах, $X^{(i)}$ – внутрішній стан після i-ого раунду, а x_i – i-тий біт стану $X^{(i)}$. Ці нерівності випливають із того, що $x_i^2 = x_i \mod 2$.

Далі розглянуто стандартні операції і оцінка степені для ARX систем, де використовуються відповідні із їх назви операції, а саме: додавання за модулем 2, додавання за модулем 2^n та циклічні зсуви. Результати подано у вигляді таблиці (2.1).

Таблиця 2.1 – Оцінка степеня d після виконання бітових операцій

Побітова операція	**Форма у $\mathbb{GF}(2)$**	**Оцінка степені**
$X \wedge Y$	xy	$d(X \wedge Y) = d(X) + d(Y)$
$X \vee Y$	$xy + x + y$	$d(X \vee Y) = d(X) + d(Y)$
$X \oplus Y$	$x + y$	$d(X \oplus Y) = \max(d(X), d(Y))$
$\neg X$	$1 + x$	$d(\neg X) = d(X)$
$X \ll n$	x	$d(X \ll n) \leq d(X)$
$X \gg n$	x	$d(X \gg n) \leq d(X)$
$X \lll n$	x	$d(X \lll n) = d(X)$
$X \ggg n$	x	$d(X \ggg n) = d(X)$

В останніх чотирьох операціях алгебраїчна стуктура полінома не змінюється, а степінь у звичайних зсувах може знизитися в результаті втрати бітів, коли довжини регістру фіксована. Що стосується додавання $X + Y$ за модулем $2^n, n \in \mathbb{Z}^+$, то воно виконується наступним чином:

$$\begin{aligned} i = 0 &: x_0 + y_0 \\ 0 < i < n &: x_i + y_i + c_{i-1},\ c_i = x_i y_i + c_{i-1} x_i + c_{i-1} y_i,\ c_0 = x_0 y_0 \end{aligned}$$

де c_i – біт переносу.

Максимальна степінь вихідних біт буде зростати від молодших біт до старших. Таким чином степінь для $0 \leq j < n$ можно оцінити так:

$$d((X+Y)_j) \geq \sum_{0<i<j} \max(d(x_i), d(y_i))$$

Для SP-мереж, де використовуються таблиці перестановок та S-блоки така оцінка теж існує. Степінь S-блоку може буде представлена наступним чином:

$$d(S(X^{(i)})) \leq |X^{(i)}| \cdot \max_{x_i \in X^{(i)}} d(x_i)$$

Як відомо, S-блоки нелінійна функція. То наприклад, якщо вона має степінь t, то після r раундів, вона може зрости до t^r.

Зрозуміло, що для атаки ефективніше знімати молодші біти та використовувати їх, як біт виходу шифратора. Крім цього для блокових шифрів також можна знімати вагу Хемінга від усього внутрішнього стану $X^{(i)}$ роботи алгоритму шифрування або окремого його байту.

Вага Хемінга визначається кількістю біт у двійковому представленні стану X. Якщо розмір такого представлення покласти за l, то $\mathrm{HW} = \sum_{i=0}^{l} x_i$, а також подається у вигляді відображення:

$$\mathrm{HW} : \{0,1\}^l \to \{0,1\}^{\lfloor \log_2 l \rfloor + 1}$$

Далі наводиться приклад коли знімається вага Хемінга від байту.

Приклад 2.1. Нехай знято окремий байт внутрішнього стану $X^{(i)}$ – $X = (x_7,x_6,x_5,x_4,x_3,x_2,x_1,x_0)$, тоді $HW(X)$ буде представлятися у вигляді чотирьохбітного значення $Y = (y_3,y_2,y_1,y_0)$.

Кожен окремий біт Y може бути обчислений наступним чином:

$$y_0 = \sum_{i=0}^{7} x_i \qquad y_2 = \sum_{0 \le i < j < k < l \le 7} x_i x_j x_k x_l$$

$$y_1 = \sum_{0 \le i < j \le 7} x_i x_j \qquad y_3 = \prod_{i=0}^{7} x_i$$

Кожен із чорирьох бітів в Y залежить від усіх бітів X, тому кожен біт може використовуватися у кубічних атаках та тестерів. Але вибір окремого з них важливий, так як обираючи більш старший біт в Y, степінь функції яка саме і представляє це значення – зростає. Тому в більшості випадків для того щоб понизити складність атаки, а саме пошук кубів, а також необхідну кількість відкритих текстів, які залежать від розміру кубу, використовують саме наймолодші біти. Чим більша розмірність кубів, тим більше відкритих текстів необхідно. Але не завжди обираючи біт із найнизчим степінем можна накопичити достатньо інформації, тому доводиться обирати і аналізувати також і інші.

Отримати степінь шифруючої функції, а також її АНФ після деякого раунду можна без таблиці істинності, як це робиться у [28]. Там знаходиться символічне представлення кожного біту для раундів 1-6, з яких можна також отримати систему лінійних виразів. Але на початкових раундах функція наймолодших вихідних бітів ще має достатньо малу кількість термів і необхіжна відносно мала частина для повного збереження її вигляду. Це доволі цікаво, але на практиці цікаво тільки для отримання інформації, яка може знадобитися якшо є деякий канал витоку.

Практична складність пошуку кубів

Нехай перебираються ітеративно усі куби розмірності i, тоді складність такого передобчислювального етапу буде складатися із перевірки суперполінома $p_{S(I)}$ на лінійність або квадратичність, з подальшим відновленням його символьного представлення. Отож загальна складність є такою:

$$S_i = C_m^i(S_{linear} + S_{quadratic} + generate(cube))$$

де

$$\begin{aligned} S_{linear} &= S_{linearTest} + \mathbb{1}_{linear}\{S_{linearRecovery}\} \\ S_{quadratic} &= S_{quadraticTest} + \mathbb{1}_{quadratic}\{S_{findSecretVariables} + S_{quadraticRecovery}\} \end{aligned}$$

складність лінійного та квадратичного аналізу відповідно.

Для квадратичних рівнянь спочатку знаходяться індекси з яких він складається, а потім перевіряється наявність кожного терму.

Далі розписується складність кожної процедури:

$$\begin{aligned} S_{linearTest} &= N_1(gen(x,y) + 2^i(4S_{f(pt)} + gen(pt))) \\ S_{linearRecovery} &= (n+1)2^i(S_{f(pt)} + gen(pt)) \\ S_{quadraticTest} &= N_2(gen(x,y,z) + 2^i(8S_{f(pt)} + gen(pt))) \\ S_{findSecretVariables} &= \kappa n 2^i(2S_{f(pt)} + gen(pt)) \\ S_{quadraticRecovery} &= 2^i(C_{c_j}^1 2S_{f(pt)} + C_{c_j}^2 4S_{f(pt)}) \end{aligned}$$

де $S_{f(x)}$ – складність шифруючої функції, pt – відкритий текст, N_1, N_2 – кількість лінійних та квадратичних тестів відповідно, κ – кількість тестів для знаходження секретних змінних, а c_j – кількість змінних з яких складається квадратичний поліном.

N_1, N_2 переважно дорівнює 100, $c_j \leq 2^6$, κ=300, та буде розглядатися 32-бітний блок відкритого тексту, та 64-бітний ключ, тобто n=64, m=32.

Можна отримати наступні приблизні середні оцінки:

$$\begin{aligned} S_{linearTest} &\leq 2^{i+9} S_f \\ S_{linearRecovery} &\leq 2^{i+6} S_f \\ S_{quadraticTest} &\leq 2^{i+10} S_f \\ S_{findSecretVariables} &\leq 2^{i+15} S_f \\ 2^{i+2} S_f \leq S_{quadraticRecovery} &\leq 2^{i+13} S_f \end{aligned}$$

$$S_i \leq C_m^i (2^{i+9} S_f + \mathbb{1}_{linear}\{2^{i+6} S_f\} + 2^{i+10} S_f + \mathbb{1}_{quadratic}\{2^{i+15} S_f + 2^{i+13} S_f\})$$

$$\Downarrow$$

$$C_m^i 2^{i+11} S_f < S_i < C_m^i 2^{i+16} S_f$$

І це лише складність перебору усіх кубів розмірності i для одного конкретно можливого біта виходу. А такий біт може обиратися серед m можливих, а також серед бітів ваги Хемінга від внутрішнього стану або конкретного його байту. Тому складність зростає ще мінімум на m.

Нехай далі будуть розгяладтися шифри із 32-бітним блоком, тобто $m = 32$. Тоді сумарна складність TC_{ca} передобчислювального етапу кубічної атаки, якщо використовувати повний перебір кубів наступна:

$$\sum_{i=1}^{32} m C_{32}^i 2^{i+16} S_f = 2^{21} S_f \sum_{i=1}^{32} C_{32}^i 2^i = 2^{21} S_f \left(\sum_{i=0}^{32} C_{32}^i 2^i - 1 \right) = 2^{21} S_f \left(3^{32} - 1\right)$$

$$2^{66} S_f < TC_{ca} < 2^{72} S_f$$

Тому важливим фактором для успішної атаки є пошук номерів бітів, які можуть давати достатню кількість незалежних рівнянь для знаходження бітів ключа.

Можна звісно знаходити також і рівняння степені > 2, але складність атаки збільшується експоненційно. Для пошуку наприклад рівнянь степені ≤ 3, складність TC_{ca} буде такою:

$$2^{67}S_f < TC_{ca} < 2^{76}S_f$$

Також це збільшує складність онлайн етапу, на якому знаходяться відповідна права часина рівнянь, а також – кількість відкритих текстів для атаки, що не є дуже гарним.

Тому для пониження складності можна використовувати наступні можливості, які будуть наведені у наступному підрозділі.

2.2 Покращення ефективності пошуку кубів

Через відсутність процедур вибору гарних кубів, необхідно якось ефективно здійснювати їх пошук, де кількість можливих кубів – 2^m.

Такими процедурами можуть бути:

– розпаралелювання;

– повторне використання таблиці істинності;

– розширення i-раундових кубів до $(i+1)$-раундових.

Розпаралелювання

Кожен куб перевіряється незалежно один від одного, завдяки цьому перебір кубів може бути легко розпаралеленим.

Нехай $\mathbb{C}$ – множина усіх кубів, $|\mathbb{C}| = 2^m$:

$$\mathbb{C} = \bigcup_{i=1}^{s} C_i,\ C_i \cap C_j = \varnothing$$

Тоді криптоаналітик, маючи s процесорів, може для кожного процесору $P_i, i = \overline{1,s}$ запустити на перевірку відповідну множину C_i. Виграш в середьому у s разів. Якщо ж вибирати для кожного P_i куби випадково, то такий шлях може не призводити до покращення часової складності, у більшості випадків це веде до збільшення десь у 2 рази.

Повторне використання таблиць істинності

Визначення 2.1. Нехай C_I – l-розмірний булевий куб. Тоді C_J – називається вкладеним кубом, якщо $J \subset I$, а C_I – обвідним.

При зростанні розмірності куба, час пошуку зростає експоненційно. Тому хотілося б зменшити часову складність. Цього можна досягти, якщо повторно використовувати значення функції обвідного куба для вкладених кубів. Для цього необхідно мати таблицю істинності для обвідного куба. Наприклад для деякого куба розмірності 25, для перевірки усіх вкладених кубів на лінійність суперполінома $p_{S(I)}$, необхідно мати 2^{33} біт пам'яті, а це 1GB. Перевіряючи один куб розмірності 25 на лінійний суперполіном, необхідно 2^{33} звернень до шифруючої функції, а для розмірності 15 та 20 – $2^{23}, 2^{28}$ відповідно. Якщо ж перевіряти усі вкладені куби розмірності 15 та 20 без повторного використання значень функції обвідного куба, необхідно зробити $C_{25}^{15} \cdot 2^{23} + C_{20}^{25} \cdot 2^{28} = 3268760 \cdot 2^{23} + 53130 \cdot 2^{28} \sim 2^{45}$ операцій, що звісно ж більше за 2^{33}. Але для повної ефективності необхідний швидкий доступ до кожного значення таблиці істинності, тому що, якщо ж доступ до

будь-якого елемента буде займати більше часу ніж повторне обчилсення значень функції, то це не має сенсу. З іншого боку, при такому підході для усіх перевіряючих кубів, значеня x,$y \in \{0,1\}^n$, які выступають у ролі ключів в тестах лінійності, будуть одні й тіж самі, що призводить до погіршення точності, тобто з'являеться підвищений ризик імовірнисних помилок у тестах лінійності, аналогічно кількісь операцій та об'єм пам'яті оцінюється і для знаходження квадратичних суперполіномів, де повторне використання може суттєво покращити часову складність, але й пам'яті потрібно набагато більше, порівняно із перевіркою і відтворенням лінійних суперполіномів.

Розширена кубічна атака

Через підвищення складності пошуку кубів із збільшенням розмірності, хотілося б мати алгоритм із відносно меншою складністю по відношеню до ітеративного перебору по усім кубам, що дозволить знаходити куби на наступному раунді. При пошуку нових кубів на $(i+1)$-ому раунді можливі варіанти доповнення вже знайдених кубів на попередньому i-тому раунді, додатковими кубічними змінними, об'єднання двох кубів або навіть зсуви індексів множини $I^{(i)}$ і подальша їх перевірка на коректність.

Зсув ліворуч та праворуч множини I на значення s буде означати наступне:

$$I^{(i)} \ll s = \{\forall i \in I^{(i)} : i - s\}$$
$$I^{(i)} \gg s = \{\forall i \in I^{(i)} : i + s\}$$

Дана техніка представлена у вигляді алгоритму (2.1) для довільного раунду $i \in (0,r]$. Спочатку атакуючий намагається знайти достатньо кубів,

щоб відновити якомога багато ключових біт на i-тому раунді. Як тільки це зроблено, увага переходить на $(i+1)$-раунд і перебираються усі можливі куби розмірностей $1 \leq k \leq u, \forall u \in \overline{2{,}32 - |I^{(i)}|}$.

Алгоритм 2.1. Розширена кубічна атака

Вхід: Множина кубів $\mathbb{C}^{(i)}$ знайдена на i-ому раунді;
u – максимальний розмір доповнюючого куба;
s – значення зсуву множини I;

Вихід: Множина $\mathbb{C}^{(i+1)}$ на $(i+1)$-ому раунді

для кожного $I_j^{(i)} \in \mathbb{C}^{(i)}$ **виконати**
  цикл *від* $k = 1$ ***до*** u **виконати**
    для кожного *куба* TI_t *розмірності* k **виконати**
      $I_j^{(i+1)} \leftarrow I_j^{(i)} \cup TI_t$;
      або $I_j^{(i+1)} \leftarrow \left(I_j^{(i)} \gg s\right) \cup TI_t$;
      якщо $p_{S(I_j^{(i+1)})}$ *пройшов тест лінійності або квадратичності* **тоді**
        якщо *якщо* $p_{S(I_j^{(i+1)})}$ *не константний* **тоді**
          $\mathbb{C}^{(i+1)} \leftarrow I_j^{(i+1)}$;
        інакше
          продовжити;
        кінець умови
      інакше
        продовжити;
      кінець умови
    кінець
  кінець циклу
кінець

2.3 Реалізація платформи

Реалізація кубічних атак та кубічних тестерів написана на мові програмування `C++` та скомпільована у середовищі `Visual Studio 2017`. CMake збірка відсутня. Код виконувався на Windows 7 x64 із чотирьох ядерним процесором `AMD A10-5750M 2,5ГГц`. Увесь код можна знайти за посиланням [22], а також у додатку Б.

У попередній роботі: «Кубічні атаки на блокові та потокові шифри», код був написаний на `C#`, який можна знайти у [21]. Він є універсальним та застовний до будь-якого шифру. На початку обираються параметри(кількість тестів лінійності, квадратичності, зашитий у чорний ящик секретний ключ та інші) та модель кубічних атак за допомогою класу *CubeAttackSettings*. У ньому представлено 6 алгоритмів шифрування: *Present, Speck, LED, IDEA, Midori, Piccolo* із різними довжинами блоків і ключів, а також тестова функція на якій можно побачити основні принципи роботи. Також присутня легка можливість розширення списку шифрів, але через таку шаблонність атаки знижується ефективність атаки.

Натомість зараз, через відсутність інтересу до аналізу шифрів із довільними довжинами, окрім 32-бітного блоку та 64-бітних ключів, реалізація атак повністю пристосована тільки до $m = 32$ і $n = 64$ з метою ефективності. Наразі платформа представляє 3 алгоритми шифрування: *Speck, Simeck, Simon*, перший з яких є ARX криптосистемою, а два інших Фейстель подібними. Але є можливість розширювати групу шифрів, наслідуючи інтерфейс *Cipher_32_64*.

Далі на рисунку (2.1) наводиться часткова структура реалізованої платформи [22].

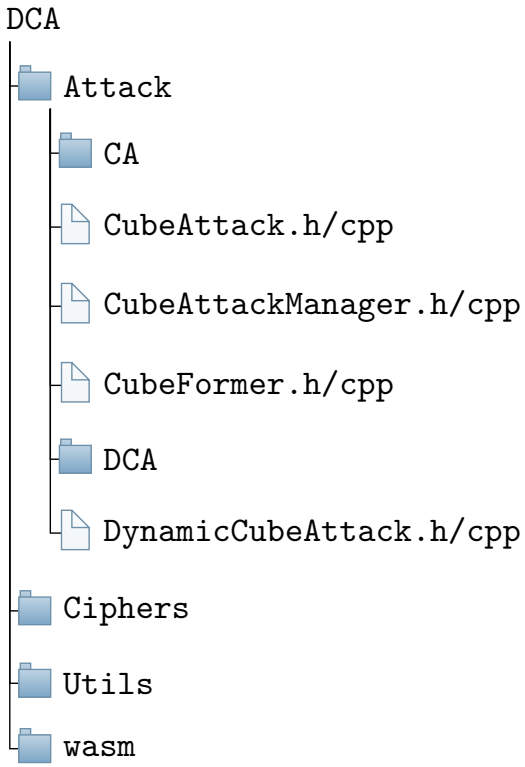

Рисунок 2.1 – Структура репозиторія DCA

Папка *CA*, що скорочено від CubeAttack містить: *CubeAttack.h/cpp, CubeAttackManager.h/cpp, CubeFormer.h/cpp*. Перший файл представляє кубічних тестерів, а також чисту кубічну атаку із пошуком лінійних та квадратичних рівнянь з подальшим знаходженням їх правої частини і отримання ключових біт, третій - стратегію вибору кубів: рандомный вибір, перебір по кубам обраної розмірності d або ітеративний по усім кубам. У першій стратегії пошук припиняється після того, як буде знайдена достатня кількість незалежних алгебраїчних рівнянь, у другій – після C_m^d ітерацій, а в останній аж після 2^m, де m відповідно дорівнює 32. Присутня також можливість легкої заміни тестів на будь-які інші, які приймають один параметр – 32-бітне значення, що є аналогом множини кубічних індексів зашитих у `uint32_t`. Вона складається із номерів бітів

параметру, які встановлені в одиницю. Другий файл містить клас, який запускає паралельно кубічні атаки для заданого масиву шифрів на максимально можливій кількості апаратних потоків.

У папці *Cipher* знаходяться реалізація раніше згаданих шифрів, з яких 2 є Фейстель-подібними шифрами та один – ARX, а у *Utils* – допоміжні функції. *DCA* у перспективі повина містити доопрацьовану версію динамічних кубічних атак згідно із [4].

У папці *wasm*, що є скороченням від *WebAssembly* знаходяться файли, які просто кажучи, необхідні для виконання коду написаного на `C/C++` у браузері.

WebAssembly[25] відносно новий бінарний формат. Формально, *WebAssembly* виконується *JavaScript*-рушієм, а не самим браузером, тому є й інші варіанти виконання, наприклад, під *NodeJS*. *WebAssembly* це просто віртуальна машина, що має пам'ять і виконуючі інструкції.

Відповідний веб-застосунок реалізованої платформи можна знайти за посиланням [23], який надає можливість тестування та перевірки наявності лінійних та квадратичних суперполіномів за обраним кубом (кубічними індексами). Для початку із списку обирається бажаний шифр, скорочений до обраної кількості раундів. Потім обирається в який спосіб знімати біт виходу шифруючої функції. Є 3 варінати, знімати конкретний біт від:

– всього вихідного стану шифрування;

– ваги Хемінга вихідного стану шифрування;

– від байту ваги Хемінга вихідного стану шифрування.

Ну і наостанок вводяться кубічні індекси, які необхідно перевірити. Деталі роботи та підказки можна беспосередньо побачити за відповідним посиланням [23].

Інтерфейс реалізованого веб-застосунку можна побачити на рисунку 2.2:

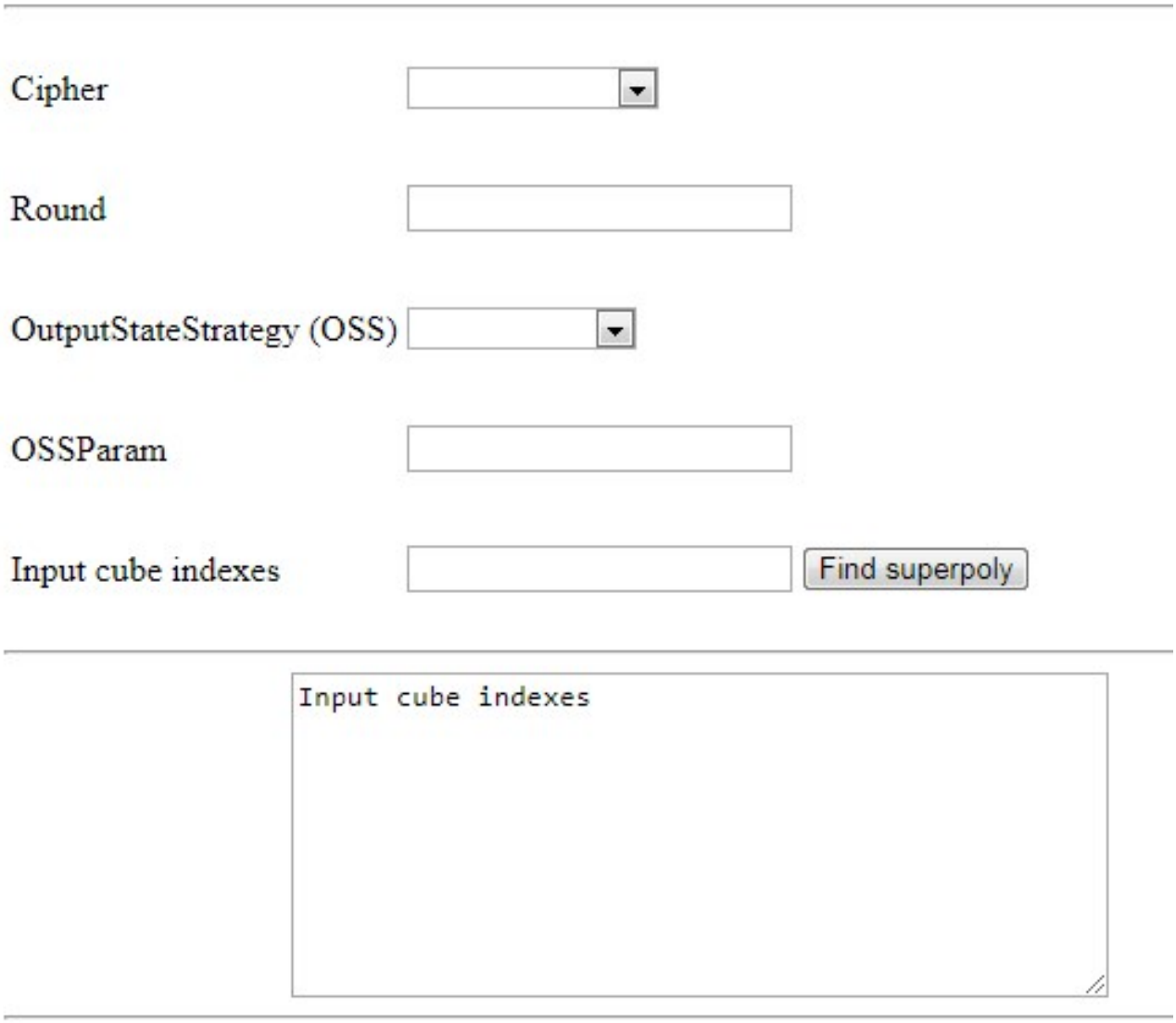

Рисунок 2.2 – Веб-застосунок кубічних тестерів

Далі на рисунку 2.3 наводиться приклад використання цього веб-застосунку. Було обрано шифр Simeck із 32-бітним блоком відкритого тексту, та 64-бітним ключем, який скорочено до чотирьох раундів. Виходом шифруючої функції приймається нульовий біт ваги Хемінга від вихідного стану шифрування. Обираючи за множину кубічних індексів $I = \{26, 29, 30, 31\}$, що еквівалентно цілочисельному представленю `0xE4000000`, можна отримати лінійний суперполіном $p_{S(I)} = k_4$.

Тож маючи 16 відкритих текстів: 0x00000000, 0x04000000, 0x20000000, 0x24000000, 0x40000000, 0x44000000, 0x60000000, 0x64000000, 0x80000000, 0x84000000, 0xA0000000, 0xA4000000, 0xC0000000, 0xC4000000, 0xE0000000 0xE4000000 із відповідними шифротекстами, можна отримати четвертий біт ключа.

Рисунок 2.3 – Приклад роботи веб-застосунку

Висновки до розділу 2

В даному розділі було проведено:

– аналіз практичного застосування кубічних атак до симетричних криптосистем;

– оцінка практичної складності кубічних атак;

– пошук покращених методів для знаходження кубів;

– розроблено та реалізовано платформу, яка дозволяє застосовувати кубічні атаки до шифрів із 32-бітним блоком відркитого тексту та 64-бітним ключем із різними методами перебору на пошуку кубів.

3 ЗАСТОСУВАТТЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ КУБІЧНИХ ТЕСТЕРІВ ТА КУБІЧНИХ АТАК ДО СИМЕТРИЧНИХ ШИФРІВ

Даний розділ присвячено огляду легковісних шифрів, а саме: Simon, Speck [15, 18], Simeck [20] та аналізу застосування кубічних тестерів та кубічних атак до них.

3.1 Опис шифрів

Simon та Speck – блокові шифри, які оптимізовано для роботи у апаратній та програмній реалізації, але якщо перший оптимізован більш до апаратної, за рахунок відмови від модульного додавання ($+$) на користь логічного і ($\odot$), то другий більш до програмної реалізації, використовуючи модульне додавання.

Simon – представляє собою Фейстель подібний шифр, а Speck належить до сімейства ARX систем. Ці два шифри мають декілька варіантів реалізації із використанням різних розмірів блоків та ключів. Блок завжди складається із двох слів, ключ може складатися із 2,3 або 4 слів, а кількість раундів залежить від розмірів блоку та ключа.

Через деякий час з'явився шифр Speck, що є аналогом шифру Simon. Він є швидшим за рахунок скорочення кількості циклічних зсувів у раундовій функції. Для Simon такими значеннями обертань є (1,8,2) то для Simeck це – (0,5,2), а також – замінивши ключовий розклад Simon'а, ключовим розкладом Speck'а (яка базується на раундовій функції).

У Simon та Simeck зберегли значення обертань для різних версій шифрів заради універсальності, жертвуючи ефективністю. У Speck же

навпаки, усі значення є однаковими окрім однієї версії шифру: 32-бітного блоку та 64-бітного ключа, у якій не захотіли збільшувати кількість раундів заради універсальності.

Операції, які використовуються у шифрах є такими:

для Simon та Simeck це:

– додавання за модулем 2, $\oplus$;

– логічне і, $\odot$;

– циклічний зсув ліворуч S^j.

а для Speck це:

– додавання за модулем 2, $\oplus$;

– додавання за модулем 2^n, $+$;

– віднімання за модулем 2^n, -;

– циклічні зсуви ліворуч та праворуч, S^j і S^{-j}, відповідно на j біт.

У шифрах задіяна раундова функція R_k, яка параметризована ключем k і виконує функцію шифрування. Для $k \in \mathbb{GF}(2)^n$ це відображення вигляду:

$$R_k : \mathbb{GF}(2)^n \times \mathbb{GF}(2)^n \to \mathbb{GF}(2)^n \times \mathbb{GF}(2)^n,$$

яка для кожного з них визначається наступним чином:

$$\begin{aligned} \text{Simon} : R_k(x,y) &= (y \oplus ((Sx \odot S^8 x) \oplus S^2 x) \oplus k, x) \\ \text{Speck} : R_k(x,y) &= ((S^{-\alpha} x + y) \oplus k, S^\beta y \oplus (S^{-\alpha} x + y) \oplus k) \\ \text{Simeck} : R_k(x,y) &= (y \oplus ((x \odot (x \lll 5)) \oplus (x \ggg 1)) \oplus k, x) \end{aligned}$$

Шифрування представляє собою композицію шифруючих перетворень, а саме раундових функцій: $R_{k_0} \circ R_{k_1} \circ \cdots \circ R_{k_{T-1}}$.

Далі будуть розглядатися і аналізуватися тільки шифри із 32-бітними блоками відкритого тексту та 64-бітні ключі і для зручночні не будуть зазначатися розмір блоку і ключа у назві шифру.

Для Speck32/64, значення параметрів обертання α та β дорівнюють 7,2 відповідно. Тому для Speck32/64 раундова функція $R_k(x,y)$ матиме

наступний вигляд:

$$R_k(x,y) = ((S^{-7}x + y) \oplus k, S^2 y \oplus (S^{-7}x + y) \oplus k)$$

Обернені раундові функції виглядають так:

$$\begin{aligned} \text{Simon} : R_k^{-1}(x,y) &= (y, x \oplus ((Sy \odot S^8 y) \oplus S^2 y) \oplus k) \\ \text{Speck} : R_k^{-1}(x,y) &= (S^7((x \oplus k) - S^{-2}(x \oplus y)), S^{-2}(x \oplus y)) \\ \text{Simeck} : R_k^{-1}(x,y) &= (y, x \oplus ((y \odot (y \ggg 5)) \oplus (y \lll 1)) \oplus k) \end{aligned}$$

Роботу раундових функцій R_k можна проілюструвати на рисунку (3.1)

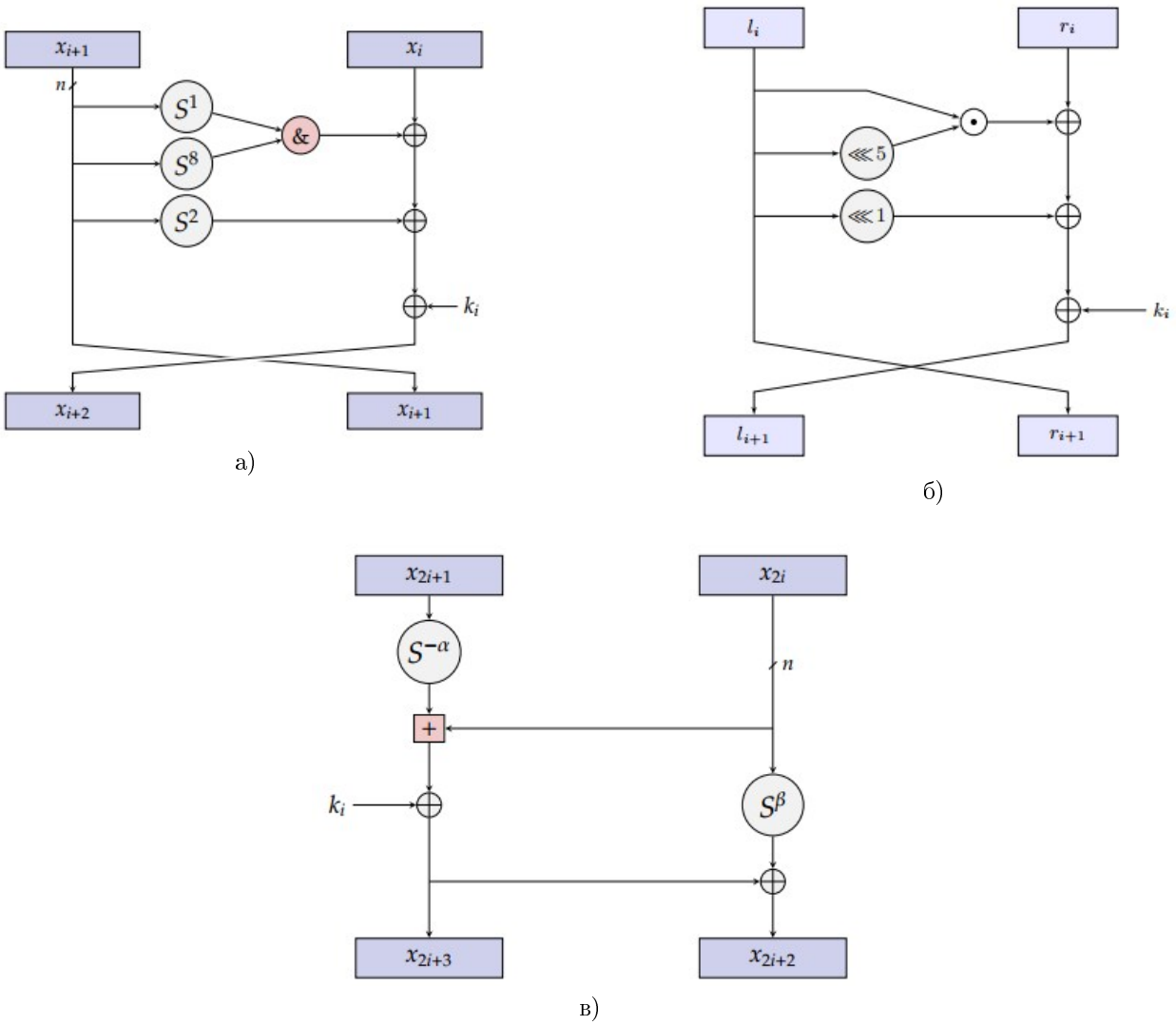

Рисунок 3.1 – Раундові функції після і-того раунду для: а) Simon, б) Simeck, в) Speck.

Кількість раундів T для Simon32/64 та Simeck32/64 дорівнюють довжині блоку відкритого тексту, а саме 32, для Speck32/64 це – 22 раунди.

Для Speck та Simeck ключевий розклад розширює початковий ключ (t_2,t_1,t_0,k_0) у $k_0,k_1,\ldots,k_{T-1}$, які використовуються на відповідному раунді. Формування раундових ключів використовує раундову функцію для генерування k_i, які розраховуються наступним чином:

– для Speck:

$$\begin{cases} t_{i+3} = (k_i + S^{-\alpha} t_i) \oplus i \\ k_{i+1} = S^{\beta} k_i \oplus t_{i+3} \end{cases}$$

– для Simeck:

$$\begin{cases} k_{i+1} = t_i \\ t_{i+3} = k_i \oplus (t_i \odot (t_i \lll 5)) \oplus (t_i \ggg 1) \oplus C \oplus (z_0)_i \end{cases}$$

Отримання значення k_i на i-тому раунді, для $0 \leq i < T$, можна побачити на рисунку 3.2, де R_p–раундова функція із параметром p який для Speck є звичайним лічильником i, а для Simeck – $C \oplus (z_0)_i$. Значення константи C визначається так: $C = 2^{32} - 4$, а $(z_0)_i$ відповідає i-ому біту послідовності z_0, яка є m-послідовністю із періодом 31 і генерується поліномом $X^5 + X^2 + 1$.

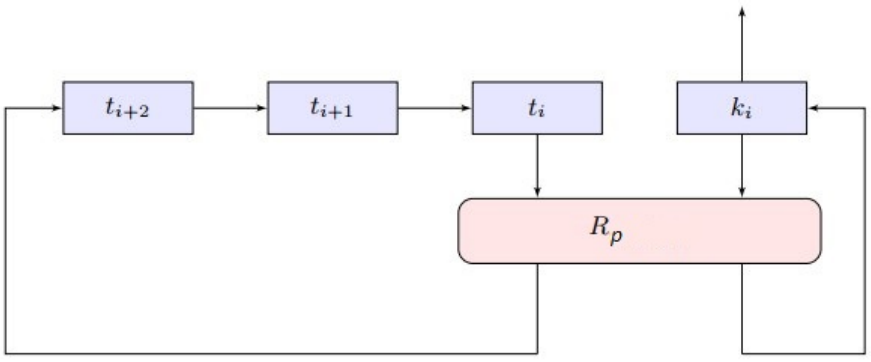

Рисунок 3.2 – Ключевий розклад Speck'а та Simeck'а

Для Simon ключем є (k_3,k_2,k_1,k_0), а раундовий ключ k_i отримується наступним чином:

$$k_{i+4} = c \oplus (z_0)_i \oplus k_i \oplus (1 \oplus S^{-1})(S^{-3}k_{i+3} \oplus k_{i+1})$$

де константа $c = 2^{32} - 4 = \texttt{0xfffc}$, а послідовність $z_0 = u = u_0u_1u_2\ldots$

$$(u)_i = (0,0,0,0,1)U^i(0,0,0,0,1)^t$$

$$U = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Downarrow$$

$$z_0 = 11111010001001010110000111001101111101000100101011000011100110$$

Ключовий розклад можна побачити на рисунку 3.3.

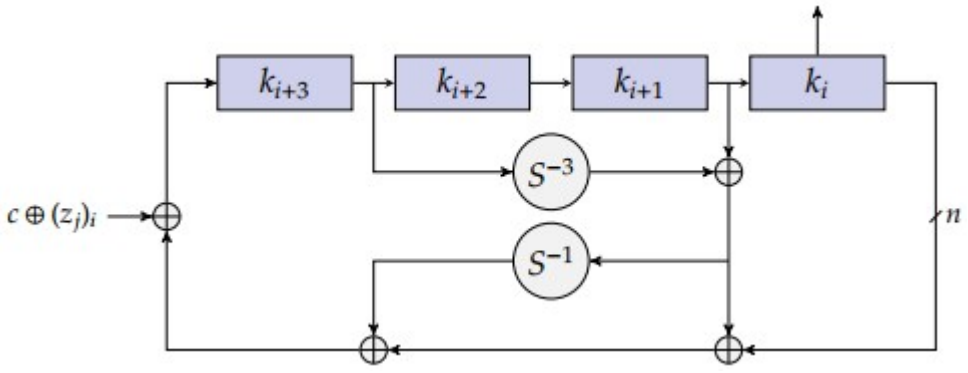

Рисунок 3.3 – Ключевий розклад Simon'a

3.2 Застосування кубічних атак

Якщо говорити про атаки на криптосистеми із моделю витоку, то звісно ж краще кубічні атаки застосовуються до легковісних Фейстель подібних шифрів (схему можна побачити на рисунку 3.4) чим, наприклад, до SPN подібних, через те, що перші використовують прості раундові функції. Також через той факт, що Фейстель подібні шифри вводять нелінійність тільки до однієї частини блоку, їм необхідно виконувати велику кількість раундів, щоб досягти однаковий запас стійкості у порівнянні із тими же SPN подібними, які використовують відносту малу кількість раундів, за рахунок перемішування і застосування перетворень до усього стану. Для першої моделі шифрів кубічні атаки дозволяють знаходити більше моделей витоку інформації, що надає більше варіантів для зламу шифру.

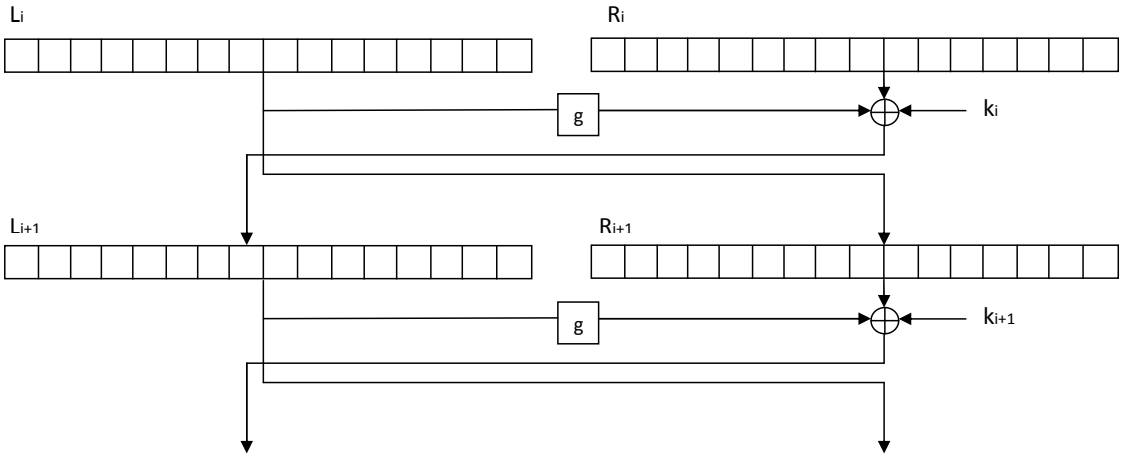

Рисунок 3.4 – Схема Фейстеля

Нумерація біт буде йти з нуля від наймолодшого біта, від LSB до MSB.

Як вже казалося для блокових шифрів у ролі знімаючого вихідного біта може виступати будь-який номер біту від 0 до $m - 1$, або вага Хемінга від проміжного стану шифрування. Здається, що краще знімати

вагу Хемінга чим окремий біт, через те, що це дає більше інформації, тобто можна знайти більше суперполіномів за менший час у порівнянні із перебором кожного біту. Таке припущення з'являється через те, що як вже було показано у підрозділі 2.1, кожен біт $HW(X)$ залежать від усіх бітів X. Наступний приклад демонструє кілька можливих варіантів коли це вірно, а коли ні.

Приклад 3.1. Припускаємо теж саме, що було у прикладі 2.1. Нехай $X = (x_7,x_6,x_5,x_4,x_3,x_2,x_1,x_0)$ окремий байт внутрішнього стану $X^{(i)}$, тоді $HW(X)$ буде представлятися у вигляді чотирьохбітного значення $Y = (y_3,y_2,y_1,y_0)$. $f_{X^{(i)})_j}$ – функція вихідного біту j стану $X^{(i)}$.

Далі наводиться кілька можливих варіантів:

1) Перший варіант:

$$\begin{aligned}
f_{X^{(i)})_0} &= \cdots \oplus v_5v_7k_0 \oplus v_0v_1k_2 \oplus v_5v_7k_1 \oplus \cdots \\
f_{X^{(i)})_1} &= \cdots \oplus v_5v_7k_{10} \oplus v_5v_7v_9k_9 \oplus \cdots \\
f_{X^{(i)})_2} &= \cdots \oplus v_5v_70 \oplus 1 \oplus \cdots \\
f_{X^{(i)})_3} &= \cdots \oplus v_5v_70 \oplus v_1 \oplus \cdots \\
f_{X^{(i)})_4} &= \cdots \oplus v_5v_70 \oplus v_1k_2 \oplus v_2 \oplus \cdots \\
f_{X^{(i)})_5} &= \cdots \oplus v_5v_70 \oplus v_0k_2 \oplus v_3 \oplus \cdots \\
f_{X^{(i)})_6} &= \cdots \oplus v_5v_70 \oplus v_7k_2 \oplus v_4 \oplus \cdots \\
f_{X^{(i)})_7} &= \cdots \oplus v_5v_7 \oplus v_0v_1k_2 \oplus v_0 \oplus \cdots
\end{aligned}$$

Тоді маючи вигляд функцій для усіх восьми бітів першого байту, можна побачити, що представлення нульового біту y_0 буде мати такий вигляд:

$$\begin{aligned}
f_{y_0} = \cdots \; &\oplus \; v_5v_7\left(k_0 \oplus k_1 \oplus k_{10} \oplus 1\right) \oplus v_5v_7v_9k_9 \oplus v_1k_2 \\
&\oplus \; v_0k_2 \oplus v_7k_2 \oplus v_0 \oplus v_1 \oplus v_2 \oplus v_3 \oplus v_4 \oplus 1 \oplus \cdots
\end{aligned}$$

У такому варіані можно бачити, що обиюраючи $I = \{5,7\}$ можна отримати лінійний суперполіном у всіх варіантах, що знімаючи конкретні біти, а саме

нульовий або перший, що – вагу Хемінга від нульового байту проміжного стану $X^{(i)}$).

2) другий варіант:

Якщо замінити $f_{X^{(i)})_1}$ на наступний варіант:

$$f_{X^{(i)})_1} = \cdots \oplus v_5 v_7 k_{10} \oplus v_5 v_7 v_9 k_9 \oplus v_5 v_7 k_{15} k_{23} \oplus \cdots$$

а всі інші функції $f_{X^{(i)})_j}$ залишити тими же самими із першого варіанту, то знімаючи у данному випадку вагу Хемінга, як у попередньому варіанті, отримати лінійний суперполіном вже не вдасться, а тільки – квадратичний.

$$\begin{aligned} f_{y_0} = \cdots \quad & \oplus \quad v_5 v_7 \left(k_0 \oplus k_1 \oplus k_{10} \oplus k_{15} k_{23} \oplus 1\right) \oplus v_5 v_7 v_9 k_9 \oplus v_1 k_2 \\ & \oplus \quad v_0 k_2 \oplus v_7 k_2 \oplus v_0 \oplus v_1 \oplus v_2 \oplus v_3 \oplus v_4 \oplus 1 \oplus \cdots \end{aligned}$$

З цих двох варіантів можна бачити, що, знімаючи вагу Хемінга, можливі такі ситуації:

– це призводить до відновлення більшого числа ключових біт, чим обираючи у ролі виходу конкретний біт;

– це може призвести до того, що велика частка мономів скоротиться і кількість відновлених ключових біт буде меньшою за варіант із зніманням одного біта;

– є можливість втрати великої частини лінійних поліномів.

3.3 Атаки на шифри

У даному підрозділі спочатку йде огляд існуючих атак на шифри, які були описані вище у підрозділі 3.1, а потім – нові пропоновані моделі кубічних атак.

Огляд існуючих атак

Описані у підрозділі 3.1 шифри відносно нові, Simon та Speck були опубліковані у 2013 році, а через 2 роки – Simeck. У таблиці 3.1 приблизно освітлені основні атаки. Під SCCA-HW у таблиці розуміється кубічні атаки за побічними каналами із використанням ваги Хемінга. Атаки за побічними каналами на представлені шифри у відкритих джерелах знайдені не були.

Таблиця 3.1 – Існуючі атаки на шифри

Шифр	**Тип атаки**	**Складність**	**# атакованих раундів**	**Джерело**
Simon32/64	Диференціальний	2^{40}	19	[30]
	Лінійний	2^{54}	16	[29]
	Динамічні кубічні атаки	$2^{57}/2^{59}$	17/22	[4]
	SCCA-HW	-	-	-
	Атаки за побічним каналам	-	-	-
Simeck32/64	Диференціальний	2^{31}	19	[31]
	Лінійний	2^{31}	13	[35]
	Кубічні атаки	-	-	-
	SCCA-HW	2^{35}	32	[33]
	Атаки за побічним каналам	-	-	-
Speck32/64	Диференціальний	2^{63}	14	[34]
	Лінійний	?	7	[35]
	Кубічні атаки	-	-	-
	SCCA-HW	-	-	-
	Атаки за побічними каналами	-	-	-

У таблиці 3.1 не освітлюється більшість атак, з котрих деякі дають навіть кращі результати за надані. На даний момент налічується приблизно 70 академічних публікацій по криптоаналізу представлених шифрів. З них можна бачити, що поки не знайдено здійснених атак на повну версію шифру, тільки на скорочені. Тому можна зробити висновок, що шифри стійкі до різних типів відомих атак. В основному, надані типи атак, які подані у таблиці 3.1 зламують шифри, які приблизно використовують $60 - 70$ відсотків раундів від основної версії шифру. (Далі буде просто казатися, що атака зламує $60 - 70\%$ шифру).

Що стосується кубічних атак, то були спроби атаки тільки на шифр Simon із 32-бітним блоком відкритого тексту та 64-бітним ключем. До нього була застосована динамічна кубічна атака, яка вперше наблизилася за ефективністю до диференціального криптоаналізу. Результатів чистої кубічної атаки та SCCA-HW до даного шифру не було знайдено.

До Simeck32/64 були знайдені результати по SCCA-HW після четвертого раунда у [33], які підпадають під сумніви. У даній роботі передобчислювальна фаза тривала кілька тижнів і відновлено 32 ключові біти, а складність знаходження ключа складає 2^{32} із використанням $\sim 2^{11.2855}$ вибраних відкритих текстів. Як висновок, ця робота позиціонує себе, як робота із найкращою атакою по побічним каналам, яка має складність істотно меншою за інші, у яких складність становить 2^{48}. Таким чином даний шифр є гарним об'єктом для аналізу.

Що стосується Speck32/64, то тут теж не було знайдено якихось правильних результатів по кубічним атаках та атакам за побічними каналами, за винятком роботи [24] у якій були знайдені помилки. Тож даний шифр також дає можливість проаналізувати шифр на кубічні атаки та отримати нові результати.

Пропоновані моделі кубічних атак

У підрозділі 3.2 було розгянуто проблему вибору моделі кубічної атаки та номера знімаючого біту. Тож усі припущення будуть використовуватися у застосуванні атак. Якщо говорити про атаки за побічними каналами, то через неправильну реалізацію шифрів у якійсь системі, атакуючий може отримувати доступ до біту інформації, що й потребується у атаці SCCA-HW. По-перше слід визначитися із номером раунду, який може дати найбільше інформації. Переважно теоретично достатньо того раунду на якому останій блок ключа завантажується у систему, але це не завжди так. Якщо номер раунду обирати таким чином, то може бути важко знайти моном/поліном із деякими ключовими бітами, для відновлення усього ключа. Тому рекомендується обирати раунд на якому досягається повна дифузія. Дифузія означає, що, якщо змінюється один біт відкритого тексту, то половина бітів у шифротексті повинна теж змінитися, аналогічно у другу сторону. Тому далі на рисунку 3.5 подано номер раунду на якому досягається ця властивість.

Wordsize	32-bit	48-bit	64-bit
Simon	7 Rounds	8 Rounds	9 Rounds
Simeck	8 Rounds	9 Rounds	11 Rounds

Рисунок 3.5 – Кількість раундів необідних для повної дифузії

Звісно ж із ростом раунду збільшується складність атаки і кількість необхідних вибраних відкритих текстів для відновлення ключа, але збільшується і кількість мономів в яких з'являються ключові біти.

Для Speck32/64 дифузія досягається $\sim$ на 5-ому раунді.

Для подальшого розгляду атак необхідно ввести кілька позначень для скорочення. Побічний канал буде позначатися через літеру L, а атака на скорочену версію шифру через – R. Моделі, які використовуються зазначені у нижньому індексі. Тобто є 3 способи знімати біт інформації: вага Хемінга від усього стану, біт ваги Хемінга від конкретного байту стану та просто біт стану. У реалізації, яка знаходиться за посиланням [22], такі моделі позначені через HW, HW_BYTE, RAW_STATE відповідно. Номер раунду після якого знімається/отримується біт інформації зазначається у верхньому індексі, а параметри у душках. Перша та третя модель мають по одному параметру, який вказує на номер індексу біта, а друга модель має два параметри, перший з яких є номером байту стану, а другий – номер біту. Тобто маємо:

– $L^r_{HW}(i), L^r_{HW\_BYTE}(n,i)$ – побічний канал у якому знімається i-ий біт ваги Хемінга/n-ого байту HW після r-ого раунду

– $R^r_{RAW\_STATE}(i)$ – атака на скорочену версію шифру до r раундів, в якій знімається i-ий біт.

Далі будуть розглядаються різні типи кубічних атак на шифри, які описані у підрозділа 3.1.

У подальших таблицях із результатами, куби будуть представлятися у вигляді беззнакового 32-ох бітного цілочисельного числа, кожен одиничний біт якого представляє наявність відповідного кубічного індексу у множині I.

Simon

Через те, що на даний шифр була надана досить ефективна кубічна атака, а саме динамічна кубічна атака [4], то для Simon не буде

розглядатися варіант атаки $R^r_{RAW\_STATE}(i)$, а тільки $L^r_{HW}(i)$ та $L^r_{HW\_BYTE}(n,i)$.

Через те, що у варіанті $L^r_{HW}(i)$ знаходження поліномів може потребувати більшу розмірність кубів у порівнянні із $L^r_{HW\_BYTE}(n,i)$, то надаватися буде якраз другий варіант, через меншу кількість необхідних вибраних відкритих текстів для відновлення ключа. Перша модель також може бути знайдена і використана, якщо це потрібно.

Для Simon'а далі буде наводитися лише деякі із знайдених моделей кубічної ааки, які дозволяються повністю відновити секретний ключ, а саме моделі: $L^7_{HW\_BYTE}(0,0)$ та $L^7_{HW\_BYTE}(1,0)$. Частина поліномів для $L^7_{HW\_BYTE}(0,0)$ із 5-ти розмірними кубами надана в додатку у таблиці А.1. Тож маючи рівняння із таблиць 3.3, 3.2, 3.4 і відповідні праві частини, можна вирішити систему та відновити усі 64 біти секретного ключа.

Таблиця 3.2 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}(1,0)$ із 5-и розмірними кубами

№	Куб	Розмірність куба	Суперполіном
1	0x10424008	5	$x_{32}x_7 \oplus x_6x_7 \oplus x_7x_{12} \oplus x_7x_{24}$
2	0x20848010	5	$x_{33}x_8 \oplus x_7x_8 \oplus x_8x_13 \oplus x_8x_{25}$
3	0x41080021	5	$x_{34}x_9 \oplus x_8x_9 \oplus x_9x_14 \oplus x_9x_{26}$
4	0x80300042	5	$x_{35}x_{10} \oplus x_9x_{10} \oplus x_{10}x_{15} \oplus x_{10}x_{27}$
5	0x210184	5	$x_0x_{11} \oplus x_{36}x_{11} \oplus x_{10}x_{11} \oplus x_{11}x_{28}$
6	0x420308	5	$x_1x_{12} \oplus x_{37}x_{12} \oplus x_{11}x_{12} \oplus x_{12}x_{29}$
7	0x40100441	5	$x_3 \oplus x_9$
8	0x40100049	5	x_{10}
9	0x8022028	5	$x_6 \oplus x_{12}$
10	0x840610	5	$x_2x_{13} \oplus x_{38}x_{13} \oplus x_{12}x_{13} \oplus x_{13}x_{30}$
11	0x900640	5	x_{13}

Таблиця 3.3 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}(0,0)$ із 6-и розмірними кубами

№	Куб	Розмірність куба	Суперполіном
1	0x800C1060	6	$1 \oplus x_3 \oplus x_7 \oplus x_{39} \oplus x_{13} \oplus x_{21} \oplus x_3x_{13} \oplus x_{13}x_{21}$
2	0x2108940	6	$1 \oplus x_{47} \oplus x_{22} \oplus x_5x_{13} \oplus x_5x_{14} \oplus x_{13}x_{23}$
3	0x580940	6	$x_{12} \oplus x_{30} \oplus x_3x_{13} \oplus x_4x_{13} \oplus x_{38}x_{13} \oplus x_{13}x_{21} \oplus x_{13}x_{55}$
4	0x100048A4	6	$1 \oplus x_{32} \oplus x_{20} \oplus x_{56} \oplus x_{25} \oplus x_{30} \oplus x_1x_8 \oplus x_3x_{12} \oplus x_6x_{13}$
5	0x1402580	6	$x_{14} \oplus x_{16} \oplus x_5x_{15} \oplus x_6x_{15} \oplus x_{40}x_{15} \oplus x_{15}x_{23} \oplus x_{15}x_{57}$
6	0x6804A00	6	$x_{15} \oplus x_{17} \oplus x_0x_6 \oplus x_0x_7 \oplus x_0x_{41} \oplus x_0x_{24} \oplus x_0x_{58}$
7	0xD009400	6	$x_{18} \oplus x_1x_7 \oplus x_1x_8 \oplus x_1x_{42} \oplus x_1x_{25} \oplus x_1x_{59}$
8	0x1A002801	6	$x_1 \oplus x_{19} \oplus x_2x_8 \oplus x_2x_9 \oplus x_2x_{43} \oplus x_2x_{26} \oplus x_2x_{60}$
9	0x14005802	6	$x_2 \oplus x_{20} \oplus x_3x_9 \oplus x_3x_{10} \oplus x_3x_{44} \oplus x_3x_{27} \oplus x_3x_{61}$

Таблиця 3.4 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}(1,0)$

№	Куб	Розмірність куба	Суперполіном
1	0x10A448	6	$1 \oplus x_8 \oplus x_{40} \oplus x_{48} \oplus x_{17} \oplus x_{22} \oplus x_{28} \oplus x_0x_9 \oplus x_4x_{11} \oplus x_5x_{14}$
2	0x40018025	6	$x_6 \oplus x_24 \oplus x_{32}x_7 \oplus x_7x_{13} \oplus x_7x_{14} \oplus x_7x_{49} \oplus x_7x_{31}$
3	0xD0004009	6	$x_4 \oplus x_{22} \oplus x_5x_{11} \oplus x_5x_{12} \oplus x_5x_{46} \oplus x_5x_{29} \oplus x_5x_{63}$
4	0x14605002	7	$x_5x_{37} \oplus x_5x_{54}$
5	0xF8000006	7	x_5
6	0xE8000406	7	x_7
7	0xE8000106	7	x_9
8	0xE8000026	7	x_{14}
⋮	⋮	⋮	⋮

Отримати усі біти ключа можна використовуючи $37 \cdot 2^5 + 11 \cdot 2^5 + 9 \cdot 2^6 + 3 \cdot 2^6 + 4 \cdot 2^7 \approx 2^{11.459}$ вибрані відкриті тексти.

Simeck

Як вже було згадано, на Simeck32/64 у 2018 році з'явилась робота, яка відновлює частину секретного ключа, а саме 32 біти із використанням $2^{11.2855}$ вибраних відкритих текстів. Тож хотілося б отримати більш вагомі результати і відновити ключ повністю. Як і для Simon'а, наводитися буде лише одна із можливих моделей атаки.

Далі наводяться таблиці 3.7, 3.5, 3.6 та одна в додатку у таблиці А.2 з використанням яких можна сформувати систему рівнянь і повністю відновити 64 біти секретного ключа.

Таблиця 3.5 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}$(1,0) із 5-и розмірними кубами

№	Куб	Розмірність куба	Суперполіном
1	0x5010A	5	$x_2x_{12} \oplus x_{12}x_{14} \oplus x_{12}x_{19}$
2	0x6020404	5	$x_{15} \oplus x_3x_{15} \oplus x_4x_{15} \oplus x_{15}x_{20}$
3	0xC040808	5	$x_0x_4 \oplus x_0x_5 \oplus x_0x_{21}$
4	0x30102020	5	$x_2 \oplus x_2x_6 \oplus x_2x_7 \oplus x_2x_{23}$
5	0x5400A	5	$x_7x_{12} \oplus x_8x_{12} \oplus x_{12}x_{24}$
6	0x810302	5	$x_{13} \oplus x_2x_{14} \oplus x_{35}x_{14} \oplus x_4x_{13} \oplus x_{14}x_{19}$
7	0x1020604	5	$x_{14} \oplus x_3x15 \oplus x_{36}x_{15} \oplus x_5x_{14} \oplus x_{15}x_{20}$
8	0x2040C08	5	$x_{15} \oplus x_0x_4 \oplus x_0x_{37} \oplus x_0x_{21} \oplus x_6x_{15}$
9	0x4081810	5	$x_0x_7 \oplus x_1x_5 \oplus x_1x_{38} \oplus x_1x_{22}$
10	0x8103020	5	$x_2x_6 \oplus x_2x_{39} \oplus x_2x_{23}$
11	0x10206040	5	$x_3x_7 \oplus x_3x_{40} \oplus x_3x_{24}$
12	0x2040C080	5	$x_4x_8 \oplus x_4x_{41} \oplus x_4x_{25}$
13	0x48008060	5	$1 \oplus x_0$
14	0x900000C1	5	$1 \oplus x_1$

Таблиця 3.6 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}$(1,0)

№	Куб	Розмірність куба	Суперполіном
1	0xC00110A	6	$x_4 \oplus x_6 \oplus x_{38} \oplus x_{20} \oplus x_{21} \oplus x_{54} \oplus x_0x_6 \oplus x_0x_{22} \oplus x_4x_{15}$
2	0xA04040A	6	x_{15}
3	0x20006809	6	$x_6 \oplus x_8 \oplus x_{40} \oplus x_{22} \oplus x_{23} \oplus x_{56} \oplus x_1x_6 \oplus x_2x_8 \oplus x_2x_{24}$
4	0x2000C0C2	6	x_4
5	0x40009412	6	$x_7 \oplus x_9 \oplus x_{41} \oplus x_{23} \oplus x_{24} \oplus x_{57} \oplus x_2x_7 \oplus x_3x_8$
6	0x90200805	6	$x_8 \oplus x_{10} \oplus x_{42} \oplus x_{24} \oplus x_{25} \oplus x_{58} \oplus x_3x_8 \oplus x_4x_9$
7	0x900060A0	6	x_3
8	0x90011081	6	x_5
9	0x90082030	6	x_9
10	0x1504A	6	$x_9 \oplus x_{11} \oplus x_{43} \oplus x_{25} \oplus x_{26} \oplus x_{59} \oplus x_4x_9 \oplus x_5x_{10}$
11	0x4600C440	7	$x_3x_{36} \oplus x_3x_7 \oplus x_3x_8 \oplus x_3x_9 \oplus x_3x_{41} \oplus x_3x_{20} \oplus x_3x_{52} \oplus x_3x_{24}$
12	0x8C008881	7	$x_4x_{37} \oplus x_4x_8 \oplus x_4x_9 \oplus x_4x_{10} \oplus x_4x_{42} \oplus x_4x_{21} \oplus x_4x_{53} \oplus x_4x_{25}$
13	0x1015032	7	$1 \oplus x_3 \oplus x_7 \oplus x_{11} \oplus x_{28} \oplus x_{33} \oplus x_{54}$
14	0x23A06	7	$x_6x_7 \oplus x_6x_{39} \oplus x_6x_{55}$
15	0x12A444	7	$1 \oplus x4 \oplus x8 \oplus x12 \oplus x29 \oplus x34 \oplus x55$
16	0x80244908	7	$1 \oplus x_{35} \oplus x_5 \oplus x_{56} \oplus x_2x_9 \oplus x_9x_{19}$
17	0x80244888	7	$1 \oplus x_5 \oplus x_9 \oplus x_{13} \oplus x_{30} \oplus x_{35} \oplus x_{56}$
18	0x20484086	7	$x_9x_{11} \oplus x_9x_{13} \oplus x_9x_{45} \oplus x_9x_{28} \oplus x_9x_{61}$
19	0x10242043	7	$x_8x_{10} \oplus x_8x_{11} \oplus x_8x_{12} \oplus x_8x44 \oplus x_8x_{27} \oplus x_8x_{60}$
20	0x81200219	7	$x_{11}x_{13} \oplus x_{11}x_{15} \oplus x_{11}x_{47} \oplus x_{11}x_{30} \oplus x_{11}x_{63}$
⋮	⋮	⋮	⋮

На перший погляд здається, що систему важко вирішити через велику кількість нелінійних рівнянь, але це не так. У таблицях міститься декілька лінійних суперполіномів за допомогою яких можна спростити майже всі рівняня, підставляючи вже знайдені біти у наступні рівняння. Тож складність знаходження усього ключа, еквівалентно складності задачі, які потребують простого шматку паперу та ручки.

Таблиця 3.7 – Приклади суперполіномів у моделі $L^7_{HW\_BYTE}(0,0)$ із 6-и розмірними кубами

№	Куб	Розмірність куба	Суперполіном
1	0x80088811	6	$x_8 \oplus x_{10} \oplus x_{42} \oplus x_{25} \oplus x_{26} \oplus x_5x_9 \oplus x_5x_{10} \oplus x_9x_{21}$
2	0x200968	6	$x_{32} \oplus x_{14} \oplus x_{48} \oplus x_{30} \oplus x_{31} \oplus x_0x_{10} \oplus x_9x_{14} \oplus x_{10}x_{16}$
3	0x11400094	6	$x_1 \oplus x_{33} \oplus x_{15} \oplus x_{16} \oplus x_{49} \oplus x_{31} \oplus x_0x_{11} \oplus x_{10}x_{15}$
4	0x802528	6	$x_2 \oplus x_{34} \oplus x_{16} \oplus x_{17} \oplus x_{50} \oplus x_0x_{11} \oplus x_1x_{12}$
5	0x5004250	6	$x_1 \oplus x_3 \oplus x_{35} \oplus x_{17} \oplus x_{18} \oplus x_{51} \oplus x_1x_{12} \oplus x_2x_{13}$
6	0x526873	6	$x_{12} \oplus x_{14} \oplus x_{46} \oplus x_{28} \oplus x_{29} \oplus x_{62} \oplus x_7x_{12} \oplus x_8x_{14} \oplus x_8x_{30}$

Отримати усі біти ключа можна використовуючи $32 \cdot 2^5 + 14 \cdot 2^5 + 6 \cdot 2^6 + 6 \cdot 2^7 + 6 \cdot 2^6 \approx 2^{11.554}$ вибрані відкриті тексти у моделі $L^7_{HW\_BYTE}(0,0)$ та $L^7_{HW\_BYTE}(1,0)$.

Якщо розглядати модель $R^r_{RAW\_STATE}(i)$ чистої розширеної кубічної атаки, то вона може наблизитися до того лінійного криптоаналізу і зламати Simeck скороченого до 13-14 раундів.

Speck

Для Speck важко знайти якісь побічні канали із використанням ваги Хемінга, через велику швидкість росту степені шифруючої функції. Також через малу кількість появи членів низької степені у порівнянні із попередніми двома шифрами. Тож кубічні атаки на скорочені варіанти даного шифру є не досить ефективними у порівнянні із іншими методами криптоаналізу.

Інші шифри

Описаний метод криптоаналізу добре застосовний до таких шифрів, як: Katan, Grain, Trivium. Через це у відкритих джерелах можна знайти застосування методу саме до цих шифрів, перший з яких є блокових, а два інші – потовими. Натомість погано застосовується до – Midori, Led, Idea, Klein, Piccolo, Rectangle, Mibs.

Висновки до розділу 3

В даному розділі на основі аналізу легковісних шифрів: Simon, Simeck, Speck, вперше до першого шифру було застосовано атаку із моделями витоку, було покращено результати атак за побічними каналами до другого шифру і оцінено стійкість третього до кубічних атак. В результаті чого, було:

1) для Simon повністю відновлено секретний ключ із використанням моделей $L^7_{HW\_BYTE}(0,0)$ та $L^7_{HW\_BYTE}(1,0)$ за наявності $2^{11.554}$ вибраних відкритих текстів;

2) для Simeck було покращено атаку за побічними каналами, яка відновклює 32 біти ключа до повного відновлення із використанням $2^{11.459}$ вибраних відкритих текстів у моделях $L^7_{HW\_BYTE}(0,0)$ та $L^7_{HW\_BYTE}(1,0)$;

3) для Speck було оцінено стійкість до кубічних атак, які виявилися неефективними до цього типу атак.

ВИСНОВКИ

У рамках виконання даної роботи були виконані такі завдання:

– було проведено аналіз, відбір та узагальнення теоретичних відомостей, необхідних для подальшого дослідження. На основі проведеного аналізу встановлено, що існуючий апарат, який представлений кубічною атакою, дозволяє отримати секретні параметри, навіть, для моделі чорного ящику за допомогою атак із вибраними відкритими текстами на блокові та потокові шифри.

Кубічні тестери ж є атаками із розпізнавачами. Перевагами їх є:

– відсутність складного та довго здійснюваного передобчислювального етапу;

– не потребують малого степеня функції.

Недоліками ж є:

– дають тільки розпізнавач, а не відновлення ключа;

– знаходить лише можливість атаки на функцію.

Якщо казати про динамічні кубічні атаки, то це більш загальна версія кубічних атак, в якій використовуються кубічні тестери для того, щоб визначити, чи є припущення стосовно підмножини ключових бітів – правильною чи ні. Та на відміну від класичної кубічної атаки, вони використовують інформацію про структуру шифру. Отже, завдяки цьому можна потенційно досягти кращих результатів. Однак є різниця, застосовуючи такую атаку на потокові та блокові шифри. Якщо у перших неможливо автоматизувати весь процес, то у других така перевага є.

Також були виконані такі завдання:

– аналіз практичного застосування кубічних атак до симетричних криптосистем;

– оцінка практичної складності кубічних атак;

– пошук покращених методів для знаходження кубів;

– розроблено та реалізовано платформу, яка дозволяє застосовувати кубічні атаки до шифрів із 32-бітним блоком відркитого тексту та 64-бітним ключем із різними методами перебору на пошуку кубів.

– на основі аналізу легковісних шифрів: Simon, Simeck, Speck, вперше до першого шифру було застосовано атаку із моделями витоку, було покращено результати атак за побічними каналами до другого шифру і оцінено стійкість третього до кубічних атак. В результаті чого, було:

1) для Simon повністю відновлено секретний ключ із використанням моделей витоку інформації після сьомого раунда, у якому знімається нульовий біт ваги Хемінга від нульового та першого байту за наявності $2^{11.554}$ вибраних відкритих текстів;

2) для Simeck було покращено атаку за побічними каналами, яка відновклює 32 біти ключа до повного відновлення із використанням $2^{11.459}$ вибраних відкритих текстів у моделях витоку інформації після сьомого раунда, у якому знімається нульовий біт ваги Хемінга від нульового та першого байту;

3) для Speck було оцінено стійкість до кубічних атак, які виявилися неефективними до цього типу атак.

ПЕРЕЛІК ПОСИЛАНЬ

1. Aumasson J-P. Cube Testers and Key Recovery Attacks On Reduced-Round MD6 and Trivium / Dinur I., Meier W., Shamir A. // In In Fast Software Encryption. Springer-Verlag – 2009. – Режим доступу: https://goo.gl/Gh4vpM.

2. Dinur I., Shamir A. Cube Attacks on Tweakable Black Box Polynomials. // In A. Joux (Ed.). Advances in Cryptology - EUROCRYPT. – 2009. – Vol. 5479. – pp. 278-299. – Режим доступу: http://eprint.iacr.org/2008/385.pdf.

3. Dinur I., Shamir A. Breaking Grain-128 with Dynamic Cube Attacks. // In Antoine Joux, editor, FSE 2011, volume 6733 of Lecture Notes in Computer Science, Springer. – 2011. – pp. 167–187. – Режим доступу: https://eprint.iacr.org/2010/570.pdf.

4. Ahmadian Z. Automated Dynamic Cube Attack on Block Ciphers: Cryptanalysis of SIMON and KATAN. / Rasoolzadeh S., Salmasizadeh M., Aref M.R // Cryptology ePrint Archive. – 2015. – Режим доступу: https://eprint.iacr.org/2015/040.pdf.

5. Salam Md Iftekhar. Investigating Cube Attacks on the Authenticated Encryption Stream Cipher ACORN / Bartlett, Harry, Dawson, Ed, Pieprzyk, Josef, Simpson, Leonie, Wong, Kenneth Koon-Ho // Cryptology ePrint Archive. – 2016 – Режим доступу: http://eprint.iacr.org/2016/743.pdf.

6. Shekh Faisal Abdul-Latip. Extended Cubes: Enhancing the Cube Attack by Extracting Low-Degree Non-Linear Equations / Mohammad Reza Reyhanitabar, Willy Susilo // School of Computer Science and Software Engineering University of Wollongong, Australia – 2011. – Режим доступу: https://goo.gl/fA35AV

7. Xinjie Zhao. Efficient Hamming weight-based side-channel cube attacks on PRESENT / Shize Guo, Fan Zhang, Tao Wang, Zhijie Shi, Huiying Liu, Keke Ji, and Jing Huang // Journal of Systems and Software. – 2013. –

№86(3). – pp. 728-743. – Режим доступу: https://goo.gl/ZRVvjr

8. Joel Lathrop. Cube attacks on cryptographic hash functions // Rochester Institute of Technology – 2009. – Режим доступу: https://goo.gl/wJQoHC

9. Gregory V. Bard. Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF (2) via SAT-Solvers / Nicolas T. Courtois, Chris Jefferson // Cryptology ePrint Archive – 2007. – Режим доступу: http://eprint.iacr.org/2007/024.pdf

10. Shekh Faisal Abdul-Latip. Algebraic and side-channel analysis of lightweigth block cipher // School of Computer Science and Software Engineering University of Wollongong, Australia – 2012. – Режим доступу: https://goo.gl/XKgC3K

11. Liren Ding. Linear Extension Cube Attack on Stream Ciphers / Yongjuan Wang, Zhufeng Li // Cryptology ePrint Archive – 2014. – Режим доступу: http://eprint.iacr.org/2014/249.pdf

12. Erfan Aghaee, Majid Rahimi, Hamed Yusefi. A Practical Iterative Side Channel Cube Attack on AES-128/256 // Cryptology ePrint Archive – 2014. – Режим доступу: ia.cr/2014/701

13. Blum M., Luby M., Rubinfeld. R. Self-Testing/Correcting with Applications to Numerical Problems. // Journal of Computer and System Sciences. – 1993. – vol 47.

14. Piotr Mroczkowski, Janusz Szmidt. The Cube Attack on Stream Cipher Trivium and Quadraticity Tests // Military Communication Institute. – 2010. – Режим доступу: http://eprint.iacr.org/2010/580.pdf

15. Beaulieu. R «The SIMON and SPECK Families of Lightweight Block Ciphers» / Shors, Douglas; Smith, Jason; Treatman-Clark, Stefan; Weeks, Bryan; Wingers, Louis // Режим доступу: http://eprint.iacr.org/2013/404.pdf.

16. Farzaneh Abed. Cryptanalysis of the Speck Family of Block Ciphers / Eik List, Stefan Lucks, and Jakob Wenzel // Cryptology ePrint Archive. – 2013 . – Режим доступу: http://eprint.iacr.org/2013/568.pdf.

17. Itai Dinur. Improved Differential Cryptanalysis of Round-Reduced Speck // Cryptology ePrint Archive. – 2014 . – Режим доступу: http://eprint.iacr.org/2014/320.

18. Ray Beaulieu. SIMON and SPECK: Block Ciphers for the Internet of Things / Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks and Louis Wingers // Cryptology ePrint Archive. – 2015 . – Режим доступу: http://eprint.iacr.org/2015/585.

19. Beaulieu R. Notes on the design and analysis of SIMON and SPECK / Shors D., Smith J., Treatman-Clark S., Weeks B., Wingers L. // Cryptology ePrint Archive. – 2017 . – Режим доступу: https://eprint.iacr.org/2017/560.

20. Yang G. The Simeck Family of Lightweight Block Ciphers / Zhu B., Suder V., Aagaard M.D., Gong G. // Cryptology ePrint Archive. – 2015 . – Режим доступу: https://eprint.iacr.org/2015/612.

21. Svichkaryov git repository – CA. // Режим доступу: https://github.com/Svichkaryov/CA.

22. Svichkaryov git repository – DCA. // Режим доступу: https://github.com/Svichkaryov/DCA.

23. Svichkaryov web-based application. // Режим доступу: https://svichkaryov.github.io/DCA/index.html.

24. Свічкарьов І.В. Кубічні атаки до блокових та потокових шифрів. – 2017.

25. WebAssembly official site. // Режим доступу: https://webassembly.org.

26. Yosuke. T. Cube Attacks on Non-Blackbox Polynomials Based on Division Property / Takanori. I, Yonglin. H, Willi. M // Cryptology ePrint Archive. – 2017 . – Режим доступу: http://eprint.iacr.org/2017/306.pdf.

27. Alon N. Testing Low-Degree Polynomials over GF(2) / Kaufman T., Krivelevich M., Litsyn S., Ron D. // Lecture Notes in Computer Science, Springer. – 2003. – Vol. 2764. – pp. 188-199. – Режим доступу: https://goo.gl/ojJqRk.

28. Peng C. Symbolic computation in block cipher with application to PRESENT / Zhu C., Zhu Y., Kang F. // Cryptology ePrint Archive. – 2012. – Режим доступу: https://eprint.iacr.org/2012/587.

29. Alizadeh J. Cryptanalysis of SIMON Variants with Connections / AlKhzaimi H, Aref M.R, Bagheri N., Gauravaram P., Kumar A., Lauridsen M.M., Sanadhya S.K. // Lecture Notes in Computer Science, Springer. – 2014. – pp. 90–107.

30. Biryukov A., Roy A., Velichkov V. Differential Analysis of Block Ciphers SIMON and SPECK // Cryptology ePrint Archive. – 2014. – Режим доступу: https://eprint.iacr.org/2014/922.

31. Kölbl S., Roy A. A Brief Comparison of Simon and Simeck // Cryptology ePrint Archive. – 2015. – Режим доступу: https://eprint.iacr.org/2015/706.

32. Bagheri N. Linear Cryptanalysis of Reduced-Round SIMECK Variants // Cryptology ePrint Archive. – 2015. – Режим доступу: https://eprint.iacr.org/2015/716.

33. Buja A.G, Faisal S. A-L., Ahmad R. A Security Analysis of IoT Encryption: Side-channel Cube Attack on Simeck32/64 // International Journal of Computer Networks and Communications (IJCNC). – 2018. – Режим доступу: http://aircconline.com/ijcnc/V10N4/10418cnc06.pdf.

34. Dinur I. Improved differential cryptanalysis of round-reduced speck. // Selected Areas in Cryptography–SAC, Springer – 2014. – pp. 147-164.

35. Ashur T., Bodden D. Linear Cryptanalysis of Reduced-Round Speck. // 2014. – Режим доступу: https://www.esat.kuleuven.be/cosic/publications/article-2666.pdf.

ДОДАТОК А ТАБЛИЦІ СУПЕРПОЛІНОМІВ

Даний додаток містить таблиці суперполіномів для шифрів Simon, Simeck, Speck, описаних у розділі 3.1.

Таблиця А.1 – Приклади суперполіномів із 5-ти розмірними кубами для Simon32/64 у моделі $L^{7}_{HW\_BYTE}$(0,0)

№	Куб	Суперполіном
1	0x88202002	$x_0x_4 \oplus x_4x_{10}$
2	0x28888	x_0x_8
3	0x400390	x_0x_{11}
4	0x428808	$x_{24} \oplus x_0x_7$
5	0x441011	x_1x_9
6	0xC0062	x_8x_{13}
7	0x1000E4	x_9x_{14}
8	0x1082022	$x_{26} \oplus x_2x_9$
9	0x1000E40	x_2x_{13}
10	0xC00620	x_1x_{12}
11	0x94004002	$x_5 \oplus x_3x_5$
12	0x90004006	x_5
13	0x88006002	x_5x_6
14	0x88082002	x_6x_{14}

Продовження таблиці А.1

№	Куб	Суперполіном
15	0x10404900	x_2
16	0x10404104	$x_1 \oplus x_{11}$
17	0x10114004	x_7x_{15}
18	0x42100840	$x_4x_{15} \oplus x_{40}x_{15} \oplus x_{14}x_{15} \oplus x_{15}x_{16}$
19	0x8402140	$x_1x_{14} \oplus x_1x_{15} \oplus x_1x_{16} \oplus x_7x_{15} \oplus x_{15}x_{25}$
20	0x22008840	$x_2x_8 \oplus x_2x_{14} \oplus x_2x_{16}$
21	0x84201080	$x_0x_5 \oplus x_0x_{41} \oplus x_0x_{15} \oplus x_0x_{17}$
22	0x10804280	$x_0x_2 \oplus x_0x_8 \oplus x_0x_{26} \oplus x_2x_{15} \oplus x_2x_{17}$
23	0x88002102	$x_0x_4 \oplus x_4x_{10} \oplus x_4x_{18}$
24	0x42000A01	$x_1x_4 \oplus x_2x_4 \oplus x_2x_{10} \oplus x_2x_{28} \oplus x_4x_{19}$
25	0x30804200	$x_1x_2 \oplus x_2x_7 \oplus x_2x_{43} \oplus x_2x_{19}$
26	0x20028408	$x_2x_6 \oplus x_6x_{12} \oplus x_6x_{20}$
27	0x21008401	$x_2x_3 \oplus x_3x_8 \oplus x_3x_{44} \oplus x_3x_{20}$
28	0x84001012	$x_5 \oplus x_2x_5 \oplus x_3x_5 \oplus x_3x_{11} \oplus x_3x_{29} \oplus x_5x_{37} \oplus x_5x_{20}$
29	0x41811	$x_3x_7 \oplus x_7x_{13} \oplus x_7x_{21}$
30	0x88002042	$x_{22} \oplus x_5x_{14}$
31	0x214404	$x_{23} \oplus x_6x_{15}$

Продовження таблиці А.1

№	Куб	Суперполіном
32	0x20029008	$x_6x_9 \oplus x_6x_{10} \oplus x_6x_{27}$
33	0x2100850	$x_5x_{13} \oplus x_{12}x_{15} \oplus x_{13}x_{15} \oplus x_{13}x_{23} \oplus x_{15}x_{30}$
34	0x42010A0	$x_0x_{13} \oplus x_0x_{14} \oplus x_0x_{31} \oplus x_6x_{14} \oplus x_{14}x_{24}$
35	0x18402100	$x_0x_1 \oplus x_1x_6 \oplus x_1x_{42} \oplus x_1x_{18}$
36	0xC2000801	$x_3x_4 \oplus x_4x_9 \oplus x_4x_{45} \oplus x_4x_{21}$
37	0x84001006	$x_4x_5 \oplus x_5x_{10} \oplus x_5x_{46} \oplus x_5x_{22}$

Таблиця А.2 – Приклади суперполіномів із 5-ти розмірними кубами для Simeck32/64 у моделі $L^7_{HW\_BYTE}$(0,0)

№	Куб	Суперполіном
1	0x2004444	$x_1 \oplus x_{13} \oplus x_1x_{13}$
2	0x4008888	$x_2 \oplus x_{14} \oplus x_2x_{14}$
3	0x22800440	x_1x_3
4	0x5000E00	x_0x_4
5	0x1002203	$x_5 \oplus x_6 \oplus x_5x_{12}$
6	0x5008A00	x_4x_6
7	0x20105020	x_7x_{10}

Продовження таблиці А.2

№	Куб	Суперполіном
8	0x4020A040	x_8x_{11}
9	0x486080	$1 \oplus x_8$
10	0x90C100	$1 \oplus x_9$
11	0x40810102	$x_{10}x_{13}$
12	0x4020A040	x_8x_{11}
13	0x80404081	x_9x_{12}
14	0x5000A40	$x_0x_4 \oplus x_4x_{15} \oplus x_4x_{16}$
15	0x50208040	$x_1 \oplus x_{33} \oplus x_{17} \oplus x_1x_{12} \oplus x_8x_{11}$
16	0x81800101	$x_{13} \oplus x_1x_{13} \oplus x_2x_{13} \oplus x_{13}x_{18}$
17	0x1204240	$x_1x_{10} \oplus x_1x_{11} \oplus x_1x_{27} \oplus x_5x_{11} \oplus x_{11}x_{22}$
18	0x4000C006	$x_4x_8 \oplus x_4x_9 \oplus x_4x_{25}$
19	0x8000800D	$x_5x_9 \oplus x_5x_{10} \oplus x_5x_{26}$
20	0x1030202	$x_6 \oplus x_6x_{10} \oplus x_6x_{11} \oplus x_6x_{27}$
21	0x2060404	$x_7 \oplus x_7x_{11} \oplus x_7x_{12} \oplus x_7x_{28}$
22	0x40C0808	$x_8 \oplus x_8x_{12} \oplus x_8x_{13} \oplus x_8x_{29}$
23	0x8181010	$x_9 \oplus x_9x_{13} \oplus x_9x_{14} \oplus x_9x_{30}$
24	0x10302020	$x_{10} \oplus x_{10}x_{14} \oplus x_{10}x_{15} \oplus x_{10}x_{31}$

Продовження таблиці А.2

№	**Куб**	**Суперполіном**
25	0x20104060	$x_{32}x_{11} \oplus x_{11}x_{15} \oplus x_{11}x_{16}$
26	0x50208040	$x_1 \oplus x_{33} \oplus x_{17} \oplus x_1x_{12} \oplus x_8x_{11}$
27	0xA0400081	$x_1 \oplus x_2 \oplus x_{34} \oplus x_{18} \oplus x_2x_{13} \oplus x_9x_{12}$
28	0x81010201	$x_5 \oplus x_5x_{12} \oplus x_6x_{10} \oplus x_6x_{43} \oplus x_6x_{27}$
29	0x2010406	$x_6 \oplus x_6x_{13} \oplus x_7x_{11} \oplus x_7x_{44} \oplus x_7x_{28}$
30	0x402080C	$x_7 \oplus x_7x_{14} \oplus x_8x_{12} \oplus x_8x_{45} \oplus x_8x_{29}$
31	0x8041018	$x_8 \oplus x_8x_{15} \oplus x_9x_{13} \oplus x_9x_{46} \oplus x_9x_{30}$
32	0x10082030	$x_{10}x_{14} \oplus x_{10}x_{47} \oplus x_{10}x_{31}$

ДОДАТОК Б ТЕКСТИ ПРОГРАМ

Даний додаток містить частково вихідний код на реалізовану платформу для застосування кубічних тестерів та атак на симетричні шифри. Повністю робочий код можна знайти за посиланням [22]. Також у [21] знаходиться платформа із попередньої роботи [24] універсальної кубічної атаки до довільних шифрів.

Лістинг файлу CubeAttack.h – файл інтерфейсу кубічних тестерів та атак

```
#pragma once
#include <vector>
#include <fstream>
#include "CubeFormer.h"
#include "../../Ciphers/Cipher_32_64.h"


enum class MAXTERM_FORM { INT_FORM, INDEX_FORM};

class CubeAttack
{
public:
        CubeAttack();
        CubeAttack(Cipher_32_64* p_cipher);
        ~CubeAttack();

        void preprocessing_phase();
        void online_phase();
        void user_mode(MAXTERM_FORM mf);

        bool linear_test(uint32_t maxterm);
        bool linear_test_blr(uint32_t maxterm);
        bool linear_test_tbt(uint32_t maxterm);
        void compute_linear_superpoly(uint32_t maxterm, uint64_t superpoly[2]);
        void print_linear_superpoly(uint32_t maxterm, const uint64_t superpoly[2], int output = -1);

        bool quadratic_test(uint32_t maxterm);
        uint64_t find_secret_variables(uint32_t maxterm);
        void compute_quadratic_superpoly(uint32_t maxterm,
                uint64_t secretVariables, std::vector<std::vector<int>>& superpoly);
        void print_quadratic_superpoly(uint32_t maxterm,
                const std::vector<std::vector<int>>& superpoly, int output = -1);

        void set_cubes(std::initializer_list<uint32_t> cubes);
        void set_extended_cubes(std::initializer_list<uint32_t> cubes, int extendedDimension);

private:
        Cipher_32_64* m_cipher;
        CubeFormer cubeFormer;

        using linear_test_t = bool(CubeAttack::*)(uint32_t);
        linear_test_t p_linearTest;

        int n_linearTest;
        int n_quadraticTest;
        int n_randSamplesForSVI;
        std::vector<uint32_t> cubesSet;

        std::ofstream m_out;
};
```

Лістинг файлу CubeAttack.cpp – файл реалізації кубічних тестерів та атак

```
#include "stdafx.h"
#include "random"
#include "sstream"
#include <thread>
#include "CubeAttack.h"
#include "../../Ciphers/Speck.h"
#include "../../Ciphers/Simeck.h"
#include "../../Ciphers/Simon.h"
#include "CubeFormer.h"


//#define ONLINE_PHASE
//#define DOUBLE_CHECK
#define LINEAR_SEARCH
#define QUADRATIC_SEARCH

//#define CONSOLE_PRINT_SUPERPOLY
#define FILE_PRINT_SUPERPOLY


CubeAttack::CubeAttack()
{
        //Speck* cipher = new Speck(OutputStateStategy::HW, 0x2);
        //Simon* cipher = new Simon(OutputStateStategy::RAW_STATE, 1);
        Simeck* cipher = new Simeck(OutputStateStategy::HW, 0x00);
        m_cipher = cipher;
        p_linearTest = &CubeAttack::linear_test_blr;
        n_linearTest = 100;
        n_quadraticTest = 100;
```

```
        n_randSamplesForSVI = 50;

#ifdef FILE_PRINT_SUPERPOLY

        std::string filePath = "Attack/CA/result/";
        std::string fileName = m_cipher->cipher_info() + std::string("_") +
                m_cipher->get_outputStateStategy_name() + std::string("_") +
                std::to_string(m_cipher->get_nBitOutput()) + ".txt";
        m_out.open(filePath + fileName);

        if (m_out.fail())
        {
                throw std::invalid_argument("Unable to open file");
        }
        m_out << "File consist cube indexes with corresponding superpoly for " <<
                m_cipher->cipher_info() << " cipher. " <<
                "Output state strategy: " << m_cipher->get_outputStateStategy_name() << ". " <<
                "Ouput bit number is: " << std::to_string(m_cipher->get_nBitOutput()) << "\n";
        m_out << "------------------------------------------------------------------------------------------\n";

#endif // FILE_PRINT_SUPERPOLY
}

CubeAttack::CubeAttack(Cipher_32_64* p_cipher)
{
        m_cipher = p_cipher;
        p_linearTest = &CubeAttack::linear_test_blr;
        n_linearTest = 100;
        n_quadraticTest = 100;
        n_randSamplesForSVI = 50;

#ifdef FILE_PRINT_SUPERPOLY

        std::string filePath = "Attack/CA/result/";
        std::string fileName = m_cipher->cipher_info() + std::string("_") +
                m_cipher->get_outputStateStategy_name() + std::string("_") +
                std::to_string(m_cipher->get_nBitOutput()) + ".txt";
        m_out.open(filePath + fileName);

        if (m_out.fail())
        {
                throw std::invalid_argument("Unable to open file");
        }
        m_out << "File consist cube indexes with corresponding superpoly for " <<
                m_cipher->cipher_info() << " cipher. " <<
                "Output state strategy: " << m_cipher->get_outputStateStategy_name() << ". " <<
                "Ouput bit number is: " << std::to_string(m_cipher->get_nBitOutput()) << "\n";
        m_out << "------------------------------------------------------------------------------------------\n";

#endif FILE_PRINT_SUPERPOLY
}

CubeAttack::~CubeAttack()
{
        if (m_out.is_open())
        {
                m_out.close();
        }
        delete m_cipher;
}

void CubeAttack::preprocessing_phase()
{
        int cubeDim = 6;
        int cubeCount = cubeFormer.get_end_flag(cubeDim);
        //uint32_t startCube = cubeFormer.get_start_cube(cubeDim);
        uint32_t startCube = cubeFormer.get_end_cube(cubeDim);
        uint32_t nextCube = startCube;

        uint64_t linear_superpoly[2];
        std::vector<std::vector<int>> quadratic_superpoly(2);
        int count = 0;
        std::vector<uint64_t> linSuperpoly = {};

        while (count != cubeCount)
        {
#ifdef LINEAR_SEARCH
                if (linear_test(nextCube))
                {
                        compute_linear_superpoly(nextCube, linear_superpoly);
                        print_linear_superpoly(nextCube, linear_superpoly);

#ifdef ONLINE_PHASE
                        if (std::find(linSuperpoly.begin(), linSuperpoly.end(), linear_superpoly[0]) ==
                                linSuperpoly.end())
                        {
                                linSuperpoly.push_back(linear_superpoly[0]);
                                cubesSet.push_back(nextCube);
                        }
#endif // ONLINE_PHASE

                }
                else
                {
#endif // LINEAR_SEARCH

#ifdef QUADRATIC_SEARCH
                        if (quadratic_test(nextCube))
                        {
                                compute_quadratic_superpoly(nextCube,
                                        find_secret_variables(nextCube), quadratic_superpoly);
                                print_quadratic_superpoly(nextCube, quadratic_superpoly);
                        }
#endif // QUADRATIC_SEARCH_F

#ifdef LINEAR_SEARCH
                }
#endif // LINEAR_SEARCH

                count++;
                if (count % 1'000'000 == 0)
                        printf("%d cube viewed for thread with id: %d\n", count, std::this_thread::get_id());

                //nextCube = cubeFormer.next_cube(nextCube);
                nextCube = cubeFormer.prev_cube(nextCube);
                //nextCube = cubeFormer.rand_cube();
                //nextCube = cubeFormer.rand_cube(5, 6);
        }
        //std::cout << "Count = " << count << std::endl;
```

```
}

void CubeAttack::online_phase()
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t key[2]        = { 0x2, 0x2 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };
        uint64_t linear_superpoly[2];
        std::vector<std::vector<int>> quadratic_superpoly;

        int output;
        int maxtermCount;
        std::vector<int> cubeIndexes = {};

        for (auto el : cubesSet)
        {
                cubeIndexes.clear();
                maxtermCount = 0;
                output = 0;
                pt = 0x0;
                for (int i = 0; i < 32; ++i)
                {
                        if (((el >> i) & 1) == 1)
                                cubeIndexes.push_back(i);
                }
                maxtermCount = cubeIndexes.size();
                uint32_t cardialDegree = 1U << maxtermCount;

                for (uint32_t k = 0; k < cardialDegree; ++k)
                {
                        for (int b = 0; b < maxtermCount; ++b)
                        {
                                if ((k & (1U << b)) > 0)
                                        pt |= (1U << cubeIndexes[b]);
                                else
                                        pt &= ~(1U << cubeIndexes[b]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, key, ciphertext);
                        output ^= m_cipher->get_bit(ciphertext);
                }

                if (linear_test(el))
                {
                        compute_linear_superpoly(el, linear_superpoly);
                        print_linear_superpoly(el, linear_superpoly, output);
                }
                else if (quadratic_test(el))
                {
                        compute_quadratic_superpoly(el,
                                find_secret_variables(el), quadratic_superpoly);
                        print_quadratic_superpoly(el, quadratic_superpoly, output);
                }
        }
}

void CubeAttack::user_mode(MAXTERM_FORM mf)
{
        char action;
        uint32_t maxterm;
        uint64_t linear_superpoly[2];
        std::vector<std::vector<int>> quadratic_superpoly;
        do
        {
                maxterm = 0x0;

                if (mf == MAXTERM_FORM::INDEX_FORM)
                {
                        uint32_t index;
                        const int end = '.'; // = 46 = 0x2E

                        printf("Input cube indexes(end with \'.\'): ");

                        for (; std::cin >> index && index != end;)
                        {
                                maxterm |= 1U << (index);
                        }

                        if (!std::cin)
                        {
                                std::cin.clear();
                                std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                                printf("Error input\n");
                        }
                }
                else if (mf == MAXTERM_FORM::INT_FORM)
                {
                        printf("Input maxterm: ");

                        std::cin >> maxterm;

                        if (!std::cin)
                        {
                                std::cin.clear();
                                std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                                printf("Error input\n");
                        }
                }

                if (linear_test(maxterm))
                {
                        std::cout << "lin\n";
                        compute_linear_superpoly(maxterm, linear_superpoly);
                        print_linear_superpoly(maxterm, linear_superpoly);
                }
                else
                {
                        if (quadratic_test(maxterm))
                        {
                                std::cout << "quadra\n";
                                compute_quadratic_superpoly(maxterm,
                                        find_secret_variables(maxterm), quadratic_superpoly);
                                print_quadratic_superpoly(maxterm, quadratic_superpoly);
                        }
                }
```

```
                printf("action(Next(n)/Exit(e)):␣");
                std::cin >> action;
        } while (action != 'e');
}

bool CubeAttack::linear_test(uint32_t maxterm)
{
        return (this->*p_linearTest)(maxterm);
}

bool CubeAttack::linear_test_blr(uint32_t maxterm)
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t x[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t y[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t xy[4]         = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };

        uint64_t rand = 0;
        std::random_device rd;
        std::mt19937_64 gen(rd());
        std::uniform_int_distribution<uint64_t> dis(0x1, 0xFFFFFFFFFFFFFFFF);

        int maxtermCount;
        std::vector<int> cubeIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
                if (((maxterm >> i) & 1) == 1)
                {
                        cubeIndexes.push_back(i);
                }
        }
        maxtermCount = cubeIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int answer = 0;
        for (int i = 0; i < n_linearTest; ++i)
        {
                //x[0] = dis(gen);
                //x[1] = dis(gen);
                //x[2] = dis(gen);
                //x[3] = dis(gen);

                //y[0] = dis(gen);
                //y[1] = dis(gen);
                //y[2] = dis(gen);
                //y[3] = dis(gen);

                rand = gen();
                x[0] = rand;
                x[1] = rand >> 16;
                x[2] = rand >> 32;
                x[3] = rand >> 48;

                rand = gen();
                y[0] = rand;
                y[1] = rand >> 16;
                y[2] = rand >> 32;
                y[3] = rand >> 48;

                xy[0] = x[0] ^ y[0];
                xy[1] = x[1] ^ y[1];
                xy[2] = x[2] ^ y[2];
                xy[3] = x[3] ^ y[3];

                for (uint32_t k = 0; k < cardialDegree; ++k)
                {
                        for (int b = 0; b < maxtermCount; ++b)
                        {
                                if ((k & (1U << b)) > 0)
                                        pt |= (1U << cubeIndexes[b]);
                                else
                                        pt &= ~(1U << cubeIndexes[b]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, x, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, y, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, xy, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, nul, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);
                }
                if (answer == 1) return false;
                answer = 0;
        }

        return true;
}

bool CubeAttack::linear_test_tbt(uint32_t maxterm)
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t x[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t y[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t z[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };
        uint64_t keyInt        = { 0x0 };
        uint64_t invKeyInt     = { 0x0 };

        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<uint16_t> dis(0x0, 0xFFFF);

        int maxtermCount = 0;
        std::vector<int> cubeIndexes = {};
        std::vector<int> keyIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
```

```
                if (((maxterm >> i) & 1) == 1)
                {
                        cubeIndexes.push_back(i);
                }
        }
        maxtermCount = cubeIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int res = 0;
        for (int k = 0; k < 64; ++k)
        {
                keyInt = 1ULL << k;
                x[0] = keyInt;
                x[1] = keyInt >> 16;
                x[2] = keyInt >> 32;
                x[3] = keyInt >> 48;

                for (uint32_t i = 0; i < cardialDegree; ++i)
                {
                        for (int j = 0; j < maxtermCount; ++j)
                        {
                                if ((i & (1U << j)) > 0)
                                        pt |= (1U << cubeIndexes[j]);
                                else
                                        pt &= ~(1U << cubeIndexes[j]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, x, ciphertext);
                        res ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, nul, ciphertext);
                        res ^= m_cipher->get_bit(ciphertext);
                }
                if (res == 1)
                        keyIndexes.push_back(k);
                res = 0;
        }

        for (auto el : keyIndexes)
        {
                for (int i = 0; i < n_linearTest; ++i)
                {
                        res = 0;

                        y[0] = dis(gen);
                        y[1] = dis(gen);
                        y[2] = dis(gen);
                        y[3] = dis(gen);

                        z[0] = y[0];
                        z[1] = y[1];
                        z[2] = y[2];
                        z[3] = y[3];

                        keyInt = 1ULL << el;
                        y[0] |= keyInt;
                        y[1] |= keyInt >> 16;
                        y[2] |= keyInt >> 32;
                        y[3] |= keyInt >> 48;

                        invKeyInt = ~keyInt;
                        z[0] &= invKeyInt;
                        z[1] &= invKeyInt >> 16;
                        z[2] &= invKeyInt >> 32;
                        z[3] &= invKeyInt >> 48;

                        for (uint32_t k = 0; k < cardialDegree; ++k)
                        {
                                for (int b = 0; b < maxtermCount; ++b)
                                {
                                        if ((k & (1U << b)) > 0)
                                                pt |= (1U << cubeIndexes[b]);
                                        else
                                                pt &= ~(1U << cubeIndexes[b]);
                                }
                                plaintext[0] = pt;
                                plaintext[1] = pt >> 16;

                                m_cipher->encrypt_block(plaintext, y, ciphertext);
                                res ^= m_cipher->get_bit(ciphertext);

                                m_cipher->encrypt_block(plaintext, z, ciphertext);
                                res ^= m_cipher->get_bit(ciphertext);
                        }
                        if (res == 0) return false;
                }
        }
        return true;
}

void CubeAttack::compute_linear_superpoly(uint32_t maxterm, uint64_t superpoly[2])
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t key[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };
        uint64_t keyInt        = { 0x0 };
        superpoly[0]           = { 0x0 };
        superpoly[1]           = { 0x0 };

        std::vector<int> cubeIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
                if (((maxterm >> i) & 1) == 1)
                {
                        cubeIndexes.push_back(i);
                }
        }
        int maxtermCount = cubeIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int constant = 0;
        int coeff = 0;

        for (uint32_t i = 0; i < cardialDegree; ++i)
        {
```

```
                for (int j = 0; j < maxtermCount; ++j)
                {
                        if ((i & (1U << j)) > 0)
                                pt |= (1U << cubeIndexes[j]);
                        else
                                pt &= ~(1U << cubeIndexes[j]);
                }
                plaintext[0] = pt;
                plaintext[1] = pt >> 16;

                m_cipher->encrypt_block(plaintext, nul, ciphertext);
                constant ^= m_cipher->get_bit(ciphertext);
        }
        superpoly[1] = constant;

        for (int k = 0; k < 64; ++k)
        {
                keyInt = 1ULL << k;
                key[0] = keyInt;
                key[1] = keyInt >> 16;
                key[2] = keyInt >> 32;
                key[3] = keyInt >> 48;

                for (uint32_t i = 0; i < cardialDegree; ++i)
                {
                        for (int j = 0; j < maxtermCount; ++j)
                        {
                                if ((i & (1U << j)) > 0)
                                        pt |= (1U << cubeIndexes[j]);
                                else
                                        pt &= ~(1U << cubeIndexes[j]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, key, ciphertext);
                        coeff ^= m_cipher->get_bit(ciphertext);
                }
                if ((constant ^ coeff) == 1)
                        superpoly[0] |= 1ULL << k;

                coeff = 0;
        }
}

void CubeAttack::print_linear_superpoly(uint32_t maxterm, const uint64_t superpoly[2], int output)
{
#ifdef DOUBLE_CHECK
        for (int i = 0; i < 10; i++)
        {
                if (!linear_test(maxterm))
                        return;
        }
#endif // DOUBLE_CHECK

        if (superpoly[0] > 0)
        {
                std::ostringstream ls;

                ls << "Cube : {";
                for (int i = 0; i < 32; ++i)
                {
                        if (((maxterm >> i) & 1) == 1)
                                ls << i << " ";
                }
                ls << "} ~ " << maxterm << "\n";

                ls << "Superpoly : " << superpoly[1];

                for (int i = 0; i < 64; ++i)
                {
                        if (((superpoly[0] >> i) & 1) == 1)
                                ls << "+x" << i;
                }
                ls << "\n";

#ifdef CONSOLE_PRINT_SUPERPOLY
                std::cout << ls.str();

                if (output != -1)
                        std::cout << "Output = " << output << std::endl;

#endif // CONSOLE_PRINT_SUPERPOLY

#ifdef FILE_PRINT_SUPERPOLY
                m_out << ls.str();

                if (output != -1)
                        m_out << "Output = " << output << "\n";
#endif // FILE_PRINT_SUPERPOLY
        }
}

bool CubeAttack::quadratic_test(uint32_t maxterm)
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t x[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t y[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t z[4]          = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t xy[4]         = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t xz[4]         = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t yz[4]         = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t xyz[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };

        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<uint16_t> dis(0x0, 0xFFFF);

        std::vector<int> cubeIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
                if (((maxterm >> i) & 1) == 1)
                {
                        cubeIndexes.push_back(i);
                }
        }
```

```
        int maxtermCount = cubeIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int answer = 0;
        for (int i = 0; i < n_quadraticTest; ++i)
        {
                x[0] = dis(gen);
                x[1] = dis(gen);
                x[2] = dis(gen);
                x[3] = dis(gen);

                y[0] = dis(gen);
                y[1] = dis(gen);
                y[2] = dis(gen);
                y[3] = dis(gen);

                z[0] = dis(gen);
                z[1] = dis(gen);
                z[2] = dis(gen);
                z[3] = dis(gen);

                xy[0] = x[0] ^ y[0];
                xy[1] = x[1] ^ y[1];
                xy[2] = x[2] ^ y[2];
                xy[3] = x[3] ^ y[3];

                xz[0] = x[0] ^ z[0];
                xz[1] = x[1] ^ z[1];
                xz[2] = x[2] ^ z[2];
                xz[3] = x[3] ^ z[3];

                yz[0] = y[0] ^ z[0];
                yz[1] = y[1] ^ z[1];
                yz[2] = y[2] ^ z[2];
                yz[3] = y[3] ^ z[3];

                xyz[0] = x[0] ^ y[0] ^ z[0];
                xyz[1] = x[1] ^ y[1] ^ z[1];
                xyz[2] = x[2] ^ y[2] ^ z[2];
                xyz[3] = x[3] ^ y[3] ^ z[3];

                for (uint32_t k = 0; k < cardialDegree; ++k)
                {
                        for (int b = 0; b < maxtermCount; ++b)
                        {
                                if ((k & (1U << b)) > 0)
                                        pt |= (1U << cubeIndexes[b]);
                                else
                                        pt &= ~(1U << cubeIndexes[b]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, x, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, y, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, z, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, xy, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, xz, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, yz, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, xyz, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, nul, ciphertext);
                        answer ^= m_cipher->get_bit(ciphertext);
                }
                if (answer == 1) return false;
                answer = 0;
        }

        return true;
}

uint64_t CubeAttack::find_secret_variables(uint32_t maxterm)
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t key[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };
        uint64_t keyInt        = { 0x0 };
        uint64_t invKey        = { 0x0 };

        uint64_t rand = 0;
        std::random_device rd;
        std::mt19937_64 gen(rd());
        std::uniform_int_distribution<uint64_t> dis(0x1, 0xFFFFFFFFFFFFFFFF);

        std::vector<int> cubeIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
                if (((maxterm >> i) & 1) == 1)
                        cubeIndexes.push_back(i);
        }
        int maxtermCount = cubeIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int output = 0;
        uint64_t secretVariablesIndexes = 0x0;

        for (int i = 0; i < 64; ++i)
        {
                keyInt = 1ULL << i;
                invKey = ~keyInt;

                for (int k = 0; k < n_randSamplesForSVI; ++k)
                {
                        rand = gen();
                        key[0] = rand;
```

```
                        key[1] = rand >> 16;
                        key[2] = rand >> 32;
                        key[3] = rand >> 48;

                        for (uint32_t j = 0; j < cardialDegree; ++j)
                        {
                                for (int b = 0; b < maxtermCount; ++b)
                                {
                                        if ((j & (1U << b)) > 0)
                                                pt |= (1U << cubeIndexes[b]);
                                        else
                                                pt &= ~(1U << cubeIndexes[b]);
                                }
                                plaintext[0] = pt;
                                plaintext[1] = pt >> 16;

                                key[0] |= keyInt;
                                key[1] |= keyInt >> 16;
                                key[2] |= keyInt >> 32;
                                key[3] |= keyInt >> 48;

                                m_cipher->encrypt_block(plaintext, key, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);

                                key[0] &= invKey;
                                key[1] &= (invKey >> 16);
                                key[2] &= (invKey >> 32);
                                key[3] &= (invKey >> 48);

                                m_cipher->encrypt_block(plaintext, key, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);
                        }

                        if (output == 1)
                        {
                                secretVariablesIndexes |= keyInt;
                                output = 0;
                                break;
                        }
                }
        }

        return secretVariablesIndexes;
}

void CubeAttack::compute_quadratic_superpoly(uint32_t maxterm,
        uint64_t secretVariables, std::vector<std::vector<int>>& superpoly)
{
        uint16_t plaintext[2]  = { 0x0, 0x0 };
        uint16_t ciphertext[2] = { 0x0, 0x0 };
        uint16_t nul[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t key[4]        = { 0x0, 0x0, 0x0, 0x0 };
        uint32_t pt            = { 0x0 };
        uint64_t keyInt        = { 0x0 };

        superpoly[0].clear();
        superpoly[1].clear();

        std::vector<int> cubeIndexes = {};
        std::vector<int> secretVariablesIndexes = {};
        for (int i = 0; i < 32; ++i)
        {
                if (((maxterm >> i) & 1) == 1)
                        cubeIndexes.push_back(i);

                if (((secretVariables >> i) & 1) == 1)
                        secretVariablesIndexes.push_back(i);

                if (((secretVariables >> (i+32)) & 1) == 1)
                        secretVariablesIndexes.push_back(i+32);
        }
        int maxtermCount = cubeIndexes.size();
        int secretVariablesCount = secretVariablesIndexes.size();
        uint32_t cardialDegree = 1U << maxtermCount;

        int output = 0;

        for (int r1 = 0; r1 < secretVariablesCount; ++r1)
        {
                keyInt = 1ULL << secretVariablesIndexes[r1];
                key[0] = keyInt;
                key[1] = keyInt >> 16;
                key[2] = keyInt >> 32;
                key[3] = keyInt >> 48;

                for (uint32_t j = 0; j < cardialDegree; ++j)
                {
                        for (int b = 0; b < maxtermCount; ++b)
                        {
                                if ((j & (1U << b)) > 0)
                                        pt |= (1U << cubeIndexes[b]);
                                else
                                        pt &= ~(1U << cubeIndexes[b]);
                        }
                        plaintext[0] = pt;
                        plaintext[1] = pt >> 16;

                        m_cipher->encrypt_block(plaintext, nul, ciphertext);
                        output ^= m_cipher->get_bit(ciphertext);

                        m_cipher->encrypt_block(plaintext, key, ciphertext);
                        output ^= m_cipher->get_bit(ciphertext);
                }

                if (output == 1)
                {
                        superpoly[0].push_back(secretVariablesIndexes[r1]);
                        superpoly[1].push_back(0);
                        output = 0;
                }
        }

        uint16_t key_01[4] = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t key_10[4] = { 0x0, 0x0, 0x0, 0x0 };
        uint16_t key_11[4] = { 0x0, 0x0, 0x0, 0x0 };

        int sVCdec = secretVariablesCount - 1;
        for (int r2_1 = 0; r2_1 < sVCdec; ++r2_1)
        {
```

```
                for (int r2_2 = r2_1 + 1; r2_2 < secretVariablesCount; ++r2_2)
                {
                        keyInt    = 1ULL << secretVariablesIndexes[r2_2];
                        key_01[0] = keyInt;
                        key_01[1] = keyInt >> 16;
                        key_01[2] = keyInt >> 32;
                        key_01[3] = keyInt >> 48;

                        keyInt    = 1ULL << secretVariablesIndexes[r2_1];
                        key_10[0] = keyInt;
                        key_10[1] = keyInt >> 16;
                        key_10[2] = keyInt >> 32;
                        key_10[3] = keyInt >> 48;

                        keyInt    = 1ULL << secretVariablesIndexes[r2_1];
                        keyInt   |= 1ULL << secretVariablesIndexes[r2_2];
                        key_11[0] = keyInt;
                        key_11[1] = keyInt >> 16;
                        key_11[2] = keyInt >> 32;
                        key_11[3] = keyInt >> 48;

                        for (uint32_t j = 0; j < cardialDegree; ++j)
                        {
                                for (int b = 0; b < maxtermCount; ++b)
                                {
                                        if ((j & (1U << b)) > 0)
                                                pt |= (1U << cubeIndexes[b]);
                                        else
                                                pt &= ~(1U << cubeIndexes[b]);
                                }
                                plaintext[0] = pt;
                                plaintext[1] = pt >> 16;

                                m_cipher->encrypt_block(plaintext, nul, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);

                                m_cipher->encrypt_block(plaintext, key_01, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);

                                m_cipher->encrypt_block(plaintext, key_10, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);

                                m_cipher->encrypt_block(plaintext, key_11, ciphertext);
                                output ^= m_cipher->get_bit(ciphertext);
                        }

                        if (output == 1)
                        {
                                superpoly[0].push_back(secretVariablesIndexes[r2_1]);
                                superpoly[1].push_back(secretVariablesIndexes[r2_2]);
                                output = 0;
                        }
                }
        }

        for (uint32_t j = 0; j < cardialDegree; ++j)
        {
                for (int b = 0; b < maxtermCount; ++b)
                {
                        if ((j & (1U << b)) > 0)
                                pt |= (1U << cubeIndexes[b]);
                        else
                                pt &= ~(1U << cubeIndexes[b]);
                }
                plaintext[0] = pt;
                plaintext[1] = pt >> 16;

                m_cipher->encrypt_block(plaintext, nul, ciphertext);
                output ^= m_cipher->get_bit(ciphertext);
        }
        superpoly[0].push_back(output);
        superpoly[1].push_back(0);
}

void CubeAttack::print_quadratic_superpoly(uint32_t maxterm,
        const std::vector<std::vector<int>>& superpoly, int output)
{
#ifdef DOUBLE_CHECK
        for (int i = 0; i < 10; i++)
        {
                if (!quadratic_test(maxterm))
                        return;
        }
#endif // DOUBLE_CHECK

        bool f_deg2 = false;
        std::ostringstream ls2;

        if (superpoly[0].size() > 0)
        {
                int superpoly0Size = superpoly[0].size() - 1;
                ls2 << "Superpoly : " << superpoly[0][superpoly0Size];

                for (int i = 0; i < superpoly0Size; ++i)
                {
                        if ((superpoly[0][i] != 0) & (superpoly[1][i] == 0))
                                ls2 << "+x" << superpoly[0][i];
                        if (superpoly[1][i] != 0)
                        {
                                f_deg2 = true;
                                ls2 << "+x" << superpoly[0][i] << "*x" << superpoly[1][i];
                        }
                }
        }

        if (superpoly[1].size() > 1 && f_deg2 == true)
        {
                std::ostringstream ls;

                ls << "Cube : {";
                for (int i = 0; i < 32; ++i)
                {
                        if (((maxterm >> i) & 1) == 1)
                                ls << i << " ";
                }
                ls << "} ~ " << maxterm << "\n";
                ls << ls2.str() << "\n";

#ifdef CONSOLE_PRINT_SUPERPOLY
```

```
                std::cout << ls.str();

                if (output != -1)
                        std::cout << "Output = " << output << std::endl;
#endif // CONSOLE_PRINT_SUPERPOLY

#ifdef FILE_PRINT_SUPERPOLY
                m_out << ls.str();

                if (output != -1)
                        m_out << "Output = " << output << "\n";
#endif // FILE_PRINT_SUPERPOLY
        }
}

void CubeAttack::set_cubes(std::initializer_list<uint32_t> cubes)
{
        cubesSet.insert(cubesSet.end(), cubes.begin(), cubes.end());
}

void CubeAttack::set_extended_cubes(std::initializer_list<uint32_t> cubes, int extendedDimension)
{
        for (auto el : cubes)
        {
                cubesSet.push_back(el);

                bool f_subCube = false;
                int count = 0;
                int cubeCount = cubeFormer.get_end_flag(extendedDimension);
                uint32_t startCube = cubeFormer.get_start_cube(extendedDimension);
                uint32_t nextCube = startCube;
                uint32_t extendedCube = el;

                while (count != cubeCount)
                {
                        for (int i = 0; i < 32; ++i)
                        {
                                if (((nextCube >> i) & 1) == 1)
                                {
                                        if (((extendedCube >> i) & 1) != 1)
                                                extendedCube |= 1U << i;
                                        else
                                                f_subCube = true;
                                }
                        }
                        count++;
                        nextCube = cubeFormer.next_cube(nextCube);
                        if (!f_subCube)
                        {
                                cubesSet.push_back(extendedCube);
                                //cubesSet.push_back(extendedCube >> 1);
                        }
                        f_subCube = false;
                        extendedCube = el;
                }
        }
}
```

Лістинг файлу CubeAttackManager.h – файл інтерфейсу менеджеру кубічних атак

```
#pragma once
#include <thread>


class CubeAttackManager
{
public:
        CubeAttackManager() = default;
        ~CubeAttackManager() = default;

        void cube_attack_run(Cipher_32_64* cipher);

        template<typename T, size_t N>
        void attack(T(&ciphers_array)[N]);

private:

};


template<typename T, size_t N>
inline void CubeAttackManager::attack(T(&ciphers_array)[N])
{
        std::vector<std::thread> threads;

        for (auto& cipher : ciphers_array)
        {
                threads.push_back(std::thread([&cipher, this] { cube_attack_run(cipher); }));
        }

        for (auto& t : threads)
        {
                t.join();
        }
}
```

Лістинг файлу CubeAttackManager.cpp – файл реалізації менеджеру кубічних атак

```
#include "stdafx.h"
#include "CubeAttack.h"
#include "CubeAttackManager.h"


void CubeAttackManager::cube_attack_run(Cipher_32_64* cipher)
{
        CubeAttack ca(cipher);
        ca.preprocessing_phase();
}
```

Лістинг файлу CubeFormer.h – файл інтерфейсу формування кубів

```
#pragma once

enum class CubeSearchStrategy { Iterate, ByDimension, Random };

class CubeFormer
{
public:
        CubeFormer() = default;
        ~CubeFormer() = default;

        uint32_t get_start_cube(int dimension);
        uint32_t next_cube(uint32_t number);
        uint32_t get_end_cube(int dimension);
        uint32_t prev_cube(uint32_t number);
        int get_end_flag(int dimension);
        uint32_t rand_cube();
        uint32_t rand_cube(int lowerDimension, int upperDimension);

private:


        uint32_t start_cubes[32] = {
                0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767,
                65535, 131071, 262143, 524287, 1048575, 2097151, 4194303, 8388607, 16777215,
                33554431, 67108863, 134217727, 268435455, 536870911, 1073741823, 2147483647
        };

        uint32_t end_cubes[32] = {
                0, 2147483648, 3221225472, 3758096384, 4026531840, 4160749568, 4227858432,
                4261412864, 4278190080, 4286578688, 4290772992, 4292870144, 4293918720,
                4294443008, 4294705152, 4294836224, 4294901760, 4294934528, 4294950912,
                4294959104, 4294965248, 4294966272, 4294966784, 4294967040, 4294967168,
                4294967232, 4294967264, 4294967280, 4294967288, 4294967292, 4294967294, 4294967295
        };

        int end_flag[32] = {
                0, 32, 496, 4960, 35960, 201376, 906142, 3365856, 10518300, 28048800,
                64512240, 129024480, 225792840, 347373600, 471435600, 565722720, 601080390,
                565722720, 471435600, 347373600, 225792840, 129024480, 64512240, 28048800,
                10518300, 3365856, 906142, 201376, 35960, 4960, 496, 32
        };

};
```

Лістинг файлу CubeFormer.cpp – файл реалізації формування кубів

```
#include "stdafx.h"
#include "CubeFormer.h"
#include "random"


uint32_t CubeFormer::get_start_cube(int dimension)
{
        return start_cubes[dimension];
}

uint32_t CubeFormer::next_cube(uint32_t number)
{
        uint32_t rightOne;
        uint32_t nextHigherOneBit;
        uint32_t rightOnesPattern;
        uint32_t next = 0;

        if (number)
        {
                rightOne          = number & -(signed)number;
                nextHigherOneBit = number + rightOne;
                rightOnesPattern = number ^ nextHigherOneBit;
                rightOnesPattern = (rightOnesPattern) / rightOne;
                rightOnesPattern >>= 2;
                next              = nextHigherOneBit | rightOnesPattern;
        }

        return next;
}

uint32_t CubeFormer::get_end_cube(int dimension)
{
        return end_cubes[dimension];
}

uint32_t CubeFormer::prev_cube(uint32_t number)
{
        return ~next_cube(~number);
}

int CubeFormer::get_end_flag(int dimension)
{
        return end_flag[dimension];
}

uint32_t CubeFormer::rand_cube()
{
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<uint32_t> dis(0x0, 0xFFFFFFFF);

        return dis(gen);
}

uint32_t CubeFormer::rand_cube(int lowerDimension, int upperDimension)
{
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<uint32_t> dis(0x0, 0xFFFFFFFF);


        return dis(gen);
}
```